AFRL-IF-WP-TM-2005-1522

A SCALABLE, RECONFIGURABLE, AND DEPENDABLE TIME-TRIGGERED ARCHITECTURE

Dr. John M. Rushby

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

JULY 2003

Final Report for 24 May 2001 – 31 January 2003

Approved for public release; distribution unlimited.

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report has been reviewed and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

//s//
DALE L. HARPER
Project Engineer
Embedded Information Systems
Engineer Branch

//s//
KENNETH LITTLEJOHN
Team Leader
Embedded Information Systems
Engineer Branch

//s//
JAMES S. WILLIAMSON, Chief
Embedded Information Systems
Information Technology Division

This report is published in the interest of scientific and technical information exchange and does not constitute approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
July 2003	Final	05/24/01 – 01/31/03

4. TITLE AND SUBTITLE
A SCALABLE, RECONFIGURABLE, AND DEPENDABLE TIME-TRIGGERED ARCHITECTURE

5a. CONTRACT NUMBER
F33615-01-C-1908

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62301E

6. AUTHOR(S)
Dr. John M. Rushby

5d. PROJECT NUMBER
L549

5e. TASK NUMBER
19

5f. WORK UNIT NUMBER
08

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER
P11396

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

Defense Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, Virginia 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TM-2005-1522

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The research performed in this project began with a study and comparison of time-triggered bus architectures for critical systems and with development of flexible methods for scheduling time-triggered systems.

Later work focused on automated synthesis and refutation (debugging) and developed a new method for bounded model checking over infinite domains based on integration of efficient decision procedures with SAT solving. The method was then extended to automated verification. Implementations of the method are available from SRI in our tools ICS and SAL.

The approach pioneered in this project for combining decision procedures with a SAT solver has now become the dominant one for bounded model checking and automated verification.

15. SUBJECT TERMS
model based design, object-oriented design, embedded information systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dale Harper
Unclassified	Unclassified	Unclassified	SAR	108	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 ext. 3588

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Part I

Introduction

This report covers the period May 24 2001 through January 31 2003 and documents work performed by SRI International for the DARPA NEST program through AFRL-WPAFB Contract F33615-01-C-1908.

Statically scheduled systems such as the time-triggered architecture (TTA) have advantages over dynamically scheduled systems in that they can achieve higher resource utilization, can more easily tolerate certain failure modes (e.g., babbling), and it is easier to provide assurance arguments for their safety and dependability. On the other hand, statically scheduled system are less flexible and less able to adapt to changing mission requirements or to a significant change in available resources. Traditionally, such adaptation had to be pre-planned and implemented as a mode change.

In this project, our original goal was to develop technology for reconfiguring time-triggered architectures during operation. This requires the ability to adapt existing schedules or to calculate new ones online: computations that used to require an overnight run must be reduced to seconds. Our initial work focussed on TTA and on fast scheduling. The first two papers below describe these aspects.

However, because there was no interest in time-triggered solutions among other program participants, nor opportunity to integrate this approach with the Open Experimental Platforms, we focussed our later work on the run-time synthesis aspects. This led to the breakthrough that we call "lazy theorem proving" which combines the fast global search of modern SAT solvers with an efficient decision procedure for a combination of important theories including linear arithmetic. At a stroke, this allows all applications of SAT solving (e.g., planning, diagnosis, bounded model checking) to be extended from models described on purely Boolean structures to those over the richer domains covered by the decision procedures.

The outputs of this research are documented in a series of technical papers that are collected in Part II of this report. Below, we provide an index and abstracts for these papers. All of them were selected for presentation at major scientific conferences, and we also provide citations for these publications.

Bus Architectures for Safety-Critical Embedded Systems by John Rushby. Published as [1].

Our initial work focussed on the Time Triggered Architecture (TTA). This paper presents a comparison of TTA with other architectures for safety-critical embedded systems.

Abstract Embedded systems for safety-critical applications often integrate multiple "functions" and must generally be fault-tolerant. These requirements lead to a need for mechanisms and services that provide protection against fault propagation and ease the construction of distributed fault-tolerant applications. A number of bus architectures have been developed to satisfy this need. This paper reviews the

requirements on these architectures, the mechanisms employed, and the services provided. Four representative architectures (SAFEbus™, SPIDER, TTA, and FlexRay) are briefly described.

On the Composition of Real-Time Schedulers by Weirong Wang and Aloysius K. Mok. Published as [2].

The disadvantage of architectures such as TTA is that they depend on pre-computed schedules. These are expensive to compute, and inflexible. This paper develops methods for constructing schedules for composite systems from those of their components and provides first steps toward more flexible static schedules.

Abstract A complex real-time embedded system may consist of multiple application components each of which has its own timeliness requirements and is scheduled by component-specific schedulers. At run-time, the schedules of the components are integrated to produce a system-level schedule of jobs to be executed. We formalize the notions of schedule composition, task group composition and component composition. Two algorithms for performing composition are proposed. The first one is an extended Earliest Deadline First algorithm which can be used as a composability test for schedules. The second algorithm, the Harmonic Component Composition algorithm (HCC) provides an online admission test for components. HCC applies a rate monotonic classification of workloads and is a hard real-time solution because responsive supply of a shared resource is guaranteed for in-budget workloads. HCC is also efficient in terms of composability and requires low computation cost for both admission control and dispatch of resources.

Lazy Theorem Proving for Bounded Model Checking over Infinite Domains by Leonardo de Moura, Harald Rueß, and Maria Sorea. Published as [3].

Our work on the run-time synthesis aspect of NEST focussed on integration of decision procedures with SAT solving. This is the seminal paper that first described the integration based on "lazy theorem proving." Facts discovered by the decision procedures are communicated to the SAT solver as additional lemmas that prune its search space. The method is "lazy" in that these lemmas are generated only as they are needed.

Abstract We investigate the combination of propositional SAT checkers with domain-specific theorem provers as a foundation for bounded model checking over infinite domains. Given a program M over an infinite state type, a linear temporal logic formula ' with domain-specific constraints over program states, and an upper bound k , our procedure determines if there is a falsifying path of length k to the hypothesis that M satisfies the specification '. This problem can be reduced to the

satisfiability of Boolean constraint formulas. Our verification engine for these kinds of formulas is lazy in that propositional abstractions of Boolean constraint formulas are incrementally refined by generating lemmas on demand from an automated analysis of spurious counterexamples using theorem proving. We exemplify bounded model checking for timed automata and for RTL level descriptions, and investigate the lazy integration of SAT solving and theorem proving.

Lemmas on Demand for Satisfiability Solvers by Leonardo de Moura and Harald Rueß. Published as [4].

Another way to look at "lazy theorem proving" is as a method for generating "lemmas on demand." This paper describes the method, and the interaction between the decision procedures and the SAT solver in more detail.

Abstract We investigate the combination of propositional SAT checkers with constraint solvers for domain-specific theories such as linear arithmetic, arrays, lists and the combination thereof. Our procedure realizes a lazy approach to satisfiability checking of propositional constraint formulas by iteratively refining Boolean formulas based on lemmas generated on demand by constraint solvers.

Embedded Deduction With ICS by Leonardo de Moura, Harald Rueß, John Rushby, and Natarajan Shankar. Published as [5].

An implementation of the method described in the previous two papers is made freely available by SRI as the tool ICS (available from `ics.csl.sri.com`). This paper describes the design decisions and capabilities of ICS.

Abstract Formal analyses can provide valuable assurance for high confidence software and systems. The analyses can range from strong typechecking through test case generation and static analysis to model checking and full verification. In all cases, the tools that support the analyses use formal deduction in some way or other. ICS is a fully automatic, high-performance decision procedure for a broad combination of theories that can be embedded in all tools of this kind to provide them with a core deductive capability of exceptional power and performance. We describe the design choices underlying ICS and the capabilities it provides.

Bounded Model Checking and Induction: From Refutation to Verification by Leonardo de Moura, Harald Rueß, and Maria Sorea. Published as [6].

One of the main applications of ICS is bounded model checking (BMC). Originally, BMC was seen as a refutational (i.e., debugging) activity, but it was soon applied to synthesis problems such as planning and test case generation. Then we discovered that variations on BMC can be used to perform verification in a very effective and

automated manner. This paper describes the methods by which BMC with ICS is extended to verification problems.

Abstract We explore the combination of bounded model checking and induction for proving safety properties of infinite-state systems. In particular, we define a general k -induction scheme and prove completeness thereof. A main characteristic of our methodology is that strengthened invariants are generated from failed k -induction proofs. This strengthening step requires quantifier-elimination, and we propose a lazy quantifier elimination procedure, which delays expensive computations of disjunctive normal forms when possible. The effectiveness of induction based on bounded model checking and invariant strengthening is demonstrated using infinite-state systems ranging from communication protocols to timed automata and (linear) hybrid automata.

Bibliography

[1] John Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[2] Weirong Wang and Aloysius K. Mok. On the composition of real-time schedulers. In *9th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Taiwan City, Taiwan, February 2003. IEEE Computer Society. Available from `http://www.cs.utexas.edu/users/weirongw/papers/rtcsa03.ps`.

[3] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002. Springer-Verlag.

[4] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Presented at SAT 2002, accepted for journal publication, May 2002. Available at `http://www.csl.sri.com/users/demoura/sat02_journal.pdf`.

[5] Leonardo de Moura, Harald Ruess, John Rushby, and Natarajan Shankar. Embedded deduction with ICS. Presented at the National Security Agency's Third High Confidence Software and Systems Conference, April 2003. Available at `http://www.csl.sri.com/~rushby/abstracts/hcss03`.

[6] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag.

Part II

Technical Papers

Presented at EMSOFT 2001: First Workshop on Embedded Software, 8–10 October 2001, Lake Tahoe CA. Springer-Verlag Lecture Notes in Computer Science, Vol. 2211, pp. 306–323.

Bus Architectures For Safety-Critical Embedded Systems*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. Embedded systems for safety-critical applications often integrate multiple "functions" and must generally be fault-tolerant. These requirements lead to a need for mechanisms and services that provide protection against fault propagation and ease the construction of distributed fault-tolerant applications. A number of bus architectures have been developed to satisfy this need. This paper reviews the requirements on these architectures, the mechanisms employed, and the services provided. Four representative architectures (SAFEbus™, SPIDER, TTA, and FlexRay) are briefly described.

1 Introduction

Embedded systems generally operate as closed-loop control systems: they repeatedly sample sensors, calculate appropriate control responses, and send those responses to actuators. In safety-critical applications, such as fly- and drive-by-wire (where there are no direct connections between the pilot and the aircraft control surfaces, nor between the driver and the car steering and brakes), requirements for ultra-high reliability demand fault tolerance and extensive redundancy. The embedded system then becomes a distributed one, and the basic control loop is complicated by mechanisms for synchronization, voting, and redundancy management.

Systems used in safety-critical applications have traditionally been *federated*, meaning that each "function" (e.g., autopilot or autothrottle in an aircraft, and brakes or suspension in a car) has its own fault-tolerant embedded control system with only minor interconnections to the systems of other functions. This provides a strong barrier to fault propagation: because the systems supporting different functions do not share resources, the failure of one function has little effect on the continued operation of others. The federated approach is expensive, however (because each function has its own replicated system), so recent applications are moving toward more integrated solutions in which some resources are shared across different functions. The new danger here is

* This research was supported by the DARPA MOBIES and NEST programs through USAF Rome Laboratory contracts F33615-00-C-1700 and F33615-01-C-1908, and by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Incorporated.

that faults may propagate from one function to another; *partitioning* is the problem of restoring to integrated systems the strong defenses against fault propagation that are naturally present in federated systems. A dual issue is that of *strong composability*: here we would like to take separately developed functions and have them run without interference on an integrated system platform with negligible integration effort.

The problems of fault tolerance, partitioning, and strong composability are challenging ones. If handled in an ad-hoc manner, their mechanisms can become the primary sources of faults and of *un*reliability in the resulting architecture [10]. Fortunately, most aspects of these problems are independent of the particular functions concerned, and they can be handled in a principled and correct manner by generic mechanisms implemented as an architecture for distributed embedded systems.

One of the essential services provided by this kind of architecture is communication of information from one distributed component to another, so a (physical or logical) communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. Consequently, these architectures are often referred to as *buses* (or *databuses*), although this term understates their complexity, sophistication, and criticality. In truth, these architectures are the safety-critical core of the applications built above them, and the choice of services to provide to those applications, and the mechanisms of their implementation, are issues of major importance in the construction and certification of safety-critical embedded systems.

In this paper, I survey some of the issues in the design of bus architectures for safety-critical embedded systems. I hope this will prove useful to potential users of these architectures and will alert others to the benefits of building on such well-considered foundations. My presentation is derived from a review of four representative architectures: two of these were primarily designed for aircraft applications and two for automobiles. The economies of scale make the automobile buses quite inexpensive—which then renders them attractive in certain aircraft applications. The aircraft buses considered are the Honeywell SAFEbus [1, 7] (the backplane dat bus used in the Boeing 777 Airplane Information Management System) and the NASA SPIDER [11] (an architecture being developed as a demonstrator for certification under the new DO254 guidelines [15]); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA) [8, 24], recently adopted by Audi and Volkswagen for automobile applications, and by Honeywell for avionics and aircraft controls functions, and FlexRay [3], which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips. A detailed comparison of these four architectures, along with more extended discussion of the issues, is available as a technical report [17].

The paper is organized as follows: Section 2 examines general issues in time-triggered systems and bus architectures, Section 3 examines the fault hypotheses under which they operate, and Section 4 describes the services that they provide; Section 5 briefly describes the four representative architectures, and conclusions are provided in Section 6.

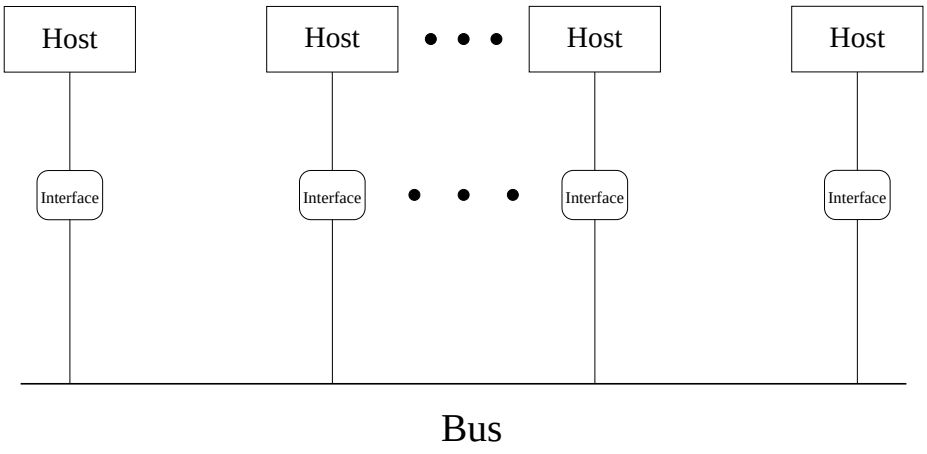

Fig. 1. Bus Interconnect

2 Time-Triggered Buses

The architectures considered here are called "buses" because multicast or broadcast communication is one of the services that they provide, and their implementations are based on a logical or physical bus. In a generic bus architecture, application programs run in *host* computers, and sensors and actuators are also connected to the hosts; an *interconnect* medium provides broadcast communications, and *interface* devices connect the hosts to the interconnect. The interfaces and interconnect comprise the bus; the combination of a host and its interface(s) is referred to as a *node*. Realizations of the interconnect may be a physical (passive) bus, as shown in Figure 1, or a centralized (active) hub, as shown in Figure 2. The interfaces may be physically proximate to the hosts, or they may form part of a more complex central hub. Many of the components will be replicated for fault tolerance.

All four of the buses considered here are primarily *time triggered*; this is a fundamental design choice that influences many aspects of their architectures and mechanisms, and sets them apart from fundamentally *event-triggered* buses such as Byteflight, CAN, Ethernet, LonWorks, or Profibus. The time-triggered and event-triggered approaches to systems design find favor in different application areas, and each has strong advocates; for integrated, safety-critical systems, however, the time-triggered approach is generally preferred. "Time triggered" means that all activities involving the bus, and often those involving components attached to the bus, are driven by the passage of time ("if it is 20 ms since the start of the frame, then read the sensor and broadcast its value"); this is distinguished from "event triggered," which means that activities are driven by the occurrence of events ("if the sensor reading changes, then broadcast its new value"). A prime contrast between these two approaches is their locus of control: a time-triggered system controls its own activity and interacts with the environment according to an internal schedule, whereas an event-triggered system is under the control of its environment and must respond to stimuli as they occur.

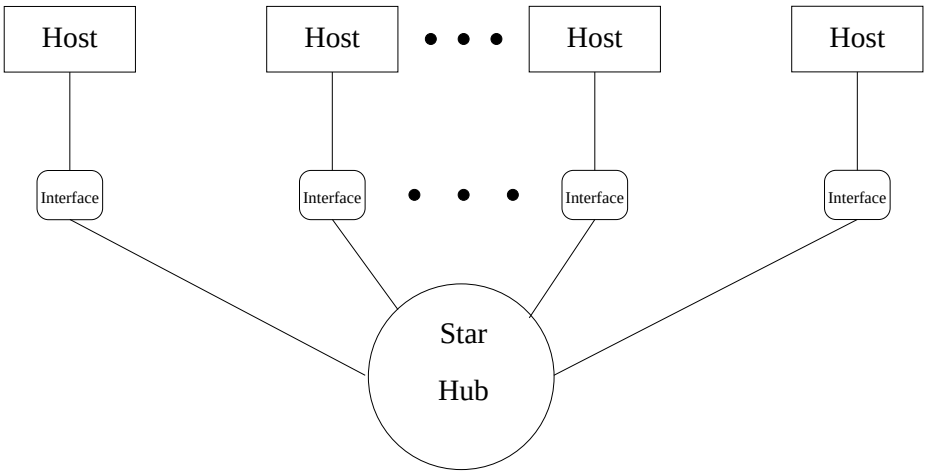

Fig. 2. Star Interconnect

Event-triggered systems allow flexible allocation of resources and this is attractive when demands are highly variable. However, in safety-critical applications it is necessary to guarantee some basic quality of service to all participants, even (or especially) in the presence of faults. Because the clients of the bus architecture are real-time embedded control systems, the required guarantees include predictable communications with low latency and low jitter (assured bandwidth is not enough). The problem with event-driven buses is that events arriving at different nodes may cause them to contend for access to the bus, so some form of media access control (i.e., a distributed mutual exclusion algorithm) is needed to ensure that each node eventually is able to transmit without interruption. The important issue is how predictable is the access achieved by each node, and how strong is the assurance that the predictions remain true in the presence of faults.

Buses such as Ethernet resolve contention probabilistically and therefore can provide only probabilistic guarantees of timely access, and no assurance at all in the presence of faults. Buses for non-safety-critical embedded systems such as CAN, LonWorks, or Profibus use various priority, preassigned slot, or token schemes to resolve contention deterministically. In CAN, for example, the message with the lowest number always wins the arbitration and therefore has to wait only for the current message to finish, while other messages must also wait for any lower-numbered messages. Thus, although contention is resolved deterministically, latency increases with load and can be bounded with only probabilistic guarantees—and these can be quite weak in the presence of faults (e.g., the current message may be retransmitted in the case of transmission failure, thereby delaying the next message, even if this has higher priority). Furthermore, faulty nodes may not adhere to expected patterns of use and may make excessive demands for service, thereby reducing that available to others. Event-triggered buses for safety-critical applications add various mechanisms to limit such demands. ARINC 629 (an avionics data bus used in the Boeing 777), for example, uses a technique some-

times referred to as “minislotting” that requires each node to wait a certain period after sending a message before it can contend to send another. Even here, however, latency is a function of load, so the Byteflight automobile protocol developed by BMW extends this mechanism with guaranteed, preallocated slots for critical messages. But even preallocated slots provide no protection against a faulty node that fails to recognize them. The worst manifestation of this kind of fault is the so-called “babbling idiot” failure where a faulty node transmits constantly, thereby compromising the operation of the entire bus.

In a time-triggered bus, there is a static preallocation of communication bandwidth in the form of a global schedule: each node knows the schedule and knows the time, and therefore knows when it is allowed to send messages, and when it should expect to receive them. Thus, contention is resolved at design time (as the schedule is constructed), when all its consequences can be examined, rather than at run time. A static schedule makes possible the control of the babbling idiot failure mode. This is achieved by interposing an independent component, called a bus *guardian*, that allows each node to transmit on the bus only when it is allowed to do so. The guardian must know when its node is allowed to access the bus, which is difficult to achieve in an event-triggered system but is conceptually simple in a time triggered system: the guardian has an independent clock and independent knowledge of the schedule and allows its node to broadcast only when indicated by the schedule.

Because all communication is triggered by the global schedule, there is no need to attach source or destination addresses to messages sent over a time-triggered bus: each node knows the sender and intended recipients of each message by virtue of the time at which it is sent. Elimination of the address fields not only reduces the size of each message, thereby greatly increasing the message bandwidth of the bus (messages are typically short in embedded control applications), but it also eliminates a potential source of serious faults: namely, the possibility that a faulty node may send messages to the wrong recipients or, worse, may masquerade as a sender other than itself.

Fault-tolerant clock synchronization is a fundamental requirement for a time-triggered bus architecture: the abstraction of a global clock is realized by each node having a local clock that is closely synchronized with the clocks of all other nodes. Tightness of the bus schedule, and hence the throughput of the bus, is strongly related to the quality of global clock synchronization that can be achieved—and this is related to the quality of the clock oscillators local to each node, and to the algorithm used to synchronize them. There are two basic classes of algorithm for clock synchronization: those based on averaging and those based on events. Averaging works by each node measuring the skew between its clock and that of each other node (e.g., by comparing the arrival time of each message with its expected value) then setting its clock to some “average” value. A simple average (e.g., the mean or median) over all clocks may be affected by wild readings from faulty clocks (which might provide different, or missing, readings to different observers), so we need a “fault-tolerant average” that is largely insensitive to a certain number of readings from faulty clocks. Schneider [18] gives a general description that applies to all averaging clock synchronization algorithms; these algorithms differ only in their choice of fault-tolerant average. The Welch-Lynch algorithm [25] is a popular choice that is characterized by use of the “fault-tolerant

midpoint" as its averaging function. Event-based algorithms rely on nodes being able to sense events directly on the interconnect: each node broadcasts a "ready" event when it is time to synchronize and sets its clock when it has seen a certain number of events from other nodes. Depending on the fault model, additional waves of "echo" or "accept" events may be needed to make this fault tolerant. The number of faulty nodes that can be tolerated, and the quality of synchronization that can be achieved, depend on the details of the algorithm, and on the fault hypothesis under which it operates. The event-based algorithm of Srikanth and Toueg [21] is particularly attractive because it achieves optimal accuracy.

3 Fault Hypotheses and Fault Containment Units

Safety-critical aerospace functions are generally required to have failure rates less than 10^{-9} per hour [5], and an architecture that is intended to support several such functions should provide assurance of failure rates better than 10^{-10} per hour. Similar requirements apply to cars (although higher rates of loss are accepted for individual cars than aircraft, there are vastly more of them, so the required failure rates are similar). Consumer-grade electronics devices have failure rates many orders of magnitude worse than this, so redundancy and fault tolerance are essential elements of a bus architecture. Redundancy may include replication of the entire bus, of the interconnect and/or the interfaces, or decomposition of those elements into smaller subcomponents that are then replicated.

Fault tolerance takes two forms in these architectures: first is that which ensures that the bus itself does not fail, second is that which eases the construction of fault-tolerant applications. Each of these mechanisms must be constructed and validated against an explicit *fault hypothesis*, and must deliver specified *services* (that may be specified to degrade in acceptable ways in the presence of faults). The fault hypothesis must describe the *modes* (i.e., kinds) of faults that are to be tolerated, and their maximum *number* and *arrival rate*. The fault hypothesis must also identify the different *fault containment units* (FCUs) in the design: these are the components that can *independently* be afflicted by faults. The division of an architecture into separate FCUs needs careful justification: there must be no propagation of faults from one FCU to another, and no "common mode failures" where a single physical event produces faults in multiple FCUs. Only physical faults (those caused by damage to, defects in, or aging of the devices employed, or by external disturbances such as cosmic rays, and electromagnetic interference) are considered in this analysis: design faults must be excluded, and must be shown to be so by stringent assurance and certification processes.

The assumption that failures of separate FCUs are independent must be ensured by careful design and assured by stringent analysis. True independence generally requires that different FCUs are served by different power supplies, and are physically and electrically isolated from each other. Providing this level of independence is expensive and it is generally undertaken only in aircraft applications. In cars, it is common to make some small compromises on independence: for example, the guardians may be fabricated on the same chip as the interface (but with their own clock oscillators), or the interface may be fabricated on the same chip as the host processor. It is necessary to

examine these compromises carefully to ensure that the loss in independence applies only to fault modes that are benign, extremely rare, or tolerated by other mechanisms.

A fault *mode* describes the kind of behavior that a faulty FCU may exhibit. The same fault may exhibit different modes at different levels of a protocol hierarchy: for example, at the electrical level, the fault mode of a faulty line driver may be that it sends an intermediate voltage (one that is neither a digital 0 nor a digital 1), while at the message level the mode of the same fault may be "Byzantine," meaning that different receivers interpret the same message in different ways (because some see the intermediate voltage as a 0, and others as a 1). Some protocols can tolerate Byzantine faults, others cannot; for those that cannot, we must show that the fault mode is controlled at the underlying electrical level.

The basic dimensions that a fault can affect are value, time, and space. A *value* fault is one that causes an incorrect value to be computed, transmitted, or received (whether as a physical voltage, a logical message, or some other representation); a *timing* fault is one that causes a value to be computed, transmitted, or received at the wrong time (whether too early, too late, or not at all); a *spatial proximity* fault is one where all matter in some specified volume is destroyed (potentially afflicting multiple FCUs). Bus-based interconnects of the kind shown in Figure 1 are vulnerable to spatial proximity faults: all redundant buses necessarily come into close proximity at each node, and general destruction in that space could sever or disrupt them all. Interconnect topologies with a central hub are far more resilient in this regard: a spatial proximity fault that destroys one or more nodes does not disrupt communication among the others (the hub may need to isolate the lines to the destroyed nodes in case these are shorted), and destruction of a hub can be tolerated if there is a duplicate in another location.

There are many ways to classify the effects of faults in any of the basic dimensions. One classification that has proved particularly effective in analysis of the types of algorithms that underlie the architectures considered here is the *hybrid* fault model of Thambidurai and Park [23]. In this classification, the effect of a fault may be *manifest*, meaning that it is reliably detected (e.g., a fault that causes an FCU to cease transmitting messages), *symmetric* meaning that whatever the effect, it is the same for all observers (e.g., an off-by-1 error), or *arbitrary*, meaning that it is entirely unconstrained. In particular, an arbitrary fault may be asymmetric or *Byzantine*, meaning that its effect is perceived differently by different observers (as in the intermediate voltage example).

The great advantage to designs that can tolerate arbitrary fault modes is that we do not have to justify assumptions about more specific fault modes: a system is shown to tolerate (say) two arbitrary faults by proving that it works in the presence of two faulty FCUs with *no assumptions whatsoever* on the behavior of the faulty components. A system that can tolerate only specific fault modes may fail if confronted by a different fault mode, so it is necessary to provide assurance that such modes cannot occur. It is this *absence* of assumptions that is the attraction, in safety-critical contexts, of systems that can tolerate arbitrary faults. This point is often misunderstood and such systems are derided as being focused on asymmetric or Byzantine faults, "which never arise in practice." Byzantine faults are just one manifestation of arbitrary behavior, and cannot simply be asserted not to occur (in fact, they have been observed in several systems that have been monitored sufficiently closely). One situation that is likely to provoke

asymmetric manifestations is a *slightly out of specification* (SOS) fault, such as the intermediate electrical voltage mentioned earlier. SOS faults in the timing dimension include those that put a signal edge very close to a clock edge, or that have signals with very slow rise and fall times (i.e., weak edges). Depending on the timing of their own clock edges, some receivers may recognize and latch such a signal, others may not, resulting in asymmetric or Byzantine behavior.

FCUs may be active (e.g., a processor) or passive (e.g., a bus); while an arbitrary-faulty active component can do anything, a passive component may change, lose, or delay data, but it cannot spontaneously create a new datum. Keyed checksums or digital signatures can sometimes be used to reduce the fault modes of an active FCU to those of a passive one. (An arbitrary-faulty active FCU can always create its own messages, but it cannot create messages purporting to come from another FCU if it does not know the key of that FCU; signatures need to be managed carefully for this reduction in fault mode to be credible.)

Any fault-tolerant architecture will fail if subjected to too many faults; generally speaking, it requires more redundancy to tolerate an arbitrary fault than a symmetric one, which in turn requires more redundancy than a manifest fault. The most effective fault-tolerant algorithms make this tradeoff automatically between number and difficulty of faults tolerated. For example, the clock synchronization algorithm of [16] can tolerate a arbitrary faults, s symmetric, and m manifest ones simultaneously provided n, the number of FCUs, satisfies $n > 3a + 2s + m$. It is provably impossible (i.e., it can be proven that no algorithm can exist) to tolerate a arbitrary faults in clock synchronization with fewer than $3a + 1$ FCUs (unless digital signatures are employed—which is equivalent to reducing the severity of the arbitrary fault mode).

Because it is algorithmically much easier to tolerate simple failure modes, some architectures (e.g., SAFEbus) arrange FCUs (the "Bus Interface Units" in the case of SAFEbus) in self-checking pairs: if the members of a pair disagree, they go offline, ensuring that the effect of their failure is seen as a manifest fault (i.e., one that is easily tolerated). Most architectures also employ substantial self-checking in each FCU; any FCU that detects a fault will shut down, thereby ensuring that its failure will be manifest. (This kind of operation is often called *fail silence*). Even with extensive self-checking and pairwise-checking, it may be possible for some fault modes to "escape," so it is generally necessary to show either that the mechanisms used have complete coverage (i.e., there will be no violation of fail silence), or to design the architecture so that it can tolerate the "escape" of at least one arbitrary fault.

Some architectures can tolerate only a single fault at a time, but can reconfigure to exclude faulty FCUs and are then able to tolerate additional faults. In such cases, the *fault arrival rate* is important: faults must not arrive faster than the architecture can reconfigure. The architectures considered here operate according to static schedules, which consist of "rounds" or "frames" that are executed repeatedly in a cyclic fashion. The acceptable fault arrival rate is often then expressed in terms of faults per round (or the inverse). It is usually important that every node is scheduled to make at least one broadcast in every round, since this is how fault status is indicated (and hence how reconfiguration is triggered).

Historical experience and analysis must be used to show that the hypothesized modes, numbers, and arrival rate are realistic, and that the architecture can indeed operate correctly under those hypotheses for its intended mission time. But sometimes things go wrong: the system may experience many simultaneous faults (e.g., from unanticipated high-intensity radiated fields (HIRF)), or other violations of its fault hypothesis. We cannot guarantee correct operation in such cases (otherwise our fault hypothesis was too conservative), but safety-critical systems generally are constructed to a "never give up" philosophy and will attempt to continue operation in a degraded mode. The usual method of operation in "never give up" mode is that each node reverts to local control of its own actuators using the best information available (e.g., each brake node applies braking force proportional to pedal pressure if it is still receiving that input, and removes all braking force if not), while at the same time attempting to regain coordination with its peers. Although it is difficult to provide assurance of correct operation during these upsets, it may be possible to provide assurance that the system returns to normal operation once the faults cease (assuming they were transients) using the ideas of self-stabilization [20].

Restart during operation may be necessary if HIRF or other environmental influences lead to violation of the fault hypothesis and cause a complete failure of the bus. Notice that this failure must be detected by the bus, and the restart must be automatic and very fast: most control systems can tolerate loss of control inputs for only a few cycles—longer outages will lead to loss of control. For example, Heiner and Thurner estimate that the maximum transient outage time for a steer-by-wire automobile application is 50ms [6].

Restart is usually initiated when an interface detects no activity on any bus line for some interval; that interface will then transmit some "wake up" message on all lines. Of course, it is possible that the interface in question is faulty (and there was bus activity all along but that interface did not detect it), or that two interfaces decide simultaneously to send the "wake up" call. The first possibility must be avoided by careful checking, preferably by independent units (e.g., both interfaces of a pair, or an interface and its guardian); the second requires some form of collision detection and resolution: this should be deterministic to guarantee an upper bound on the time to reach resolution (that will allow a single interface can send an uninterrupted "wake up" message) and, ideally, should not depend on collision detection (because this cannot be done reliably). Notice that it must be possible to perform startup and restart reliably even in the presence of faulty components.

4 Services

The essential basic purpose of these bus architectures is to make it *possible* to build reliable distributed applications; a desirable purpose is to make it *straightforward* to build such applications. The basic services provided by the bus architectures considered here comprise clock synchronization, time-triggered activation, and reliable message delivery. Some of the architectures provide additional services; their purpose is to assist straightforward construction of reliable distributed applications by providing these services in an application-independent manner, thereby relieving the applications

of the need to implement these capabilities themselves. Not only does this simplify the construction of application software, it is sometimes possible to provide *better* services when these are implemented at the architecture level, and it is also possible to provide strong assurance that they are implemented correctly.

Applications that perform safety-critical functions must generally be replicated for fault tolerance. There are many ways to organize fault-tolerant replicated computations, but a basic distinction is between those that use *exact* agreement, and those that use *approximate* agreement. Systems that use approximate agreement generally run several copies of the application in different nodes, each using its own sensors, with little coordination across the different nodes. The motivation for this is a "folk belief" that it promotes fault tolerance: coordination is believed to introduce the potential for common mode failures. Because different sensors cannot be expected to deliver exactly the same readings, the outputs (i.e., actuator commands) computed in the different nodes will also differ. Thus, the only way to detect faulty outputs is by looking for values that differ by "a lot" from the others. Hence, these systems use some form of selection or threshold voting to select a good value to send to the actuators, and similar techniques to identify faulty nodes that should be excluded from the configuration. A difficulty for applications of the kind considered here is that hosts accumulate state that diverges from that of others over time (e.g., velocity and position as a result of integrating acceleration), and they execute mode switches that are discrete decisions based on local sensor values (e.g., change the gain schedule in the control laws if the altitude, or temperature, is above a specific value). Thus small differences in sensor readings can lead to major differences in outputs and this can mislead the approximate selection or voting mechanisms into choosing a faulty value, or excluding a nonfaulty node. The fix to these problems is to attempt to coordinate discrete mode switches and periodically to bring state data into convergence. But these fixes are highly application specific, and they are contrary to the original philosophy that motivated the choice of approximate agreement—hence, there is a good chance of doing them wrong. There are numerous examples that justify this concern; several that were discovered in flight tests are documented by Mackall and colleagues [10]. The essential points of Mackall's data is that all the failures observed in flight test were due to bugs in the design of the fault tolerance mechanisms themselves, and all these bugs could be traced to difficulties in organizing and coordinating systems based on approximate agreement.

Systems based on exact agreement face up to the fact that coordination among replicated computations is necessary, and they take the necessary steps to do it right. If we are to use exact agreement, then every replica must perform the same computation on the same data: any disagreement on the outputs then indicates a fault; comparison can be used to detect those faults, and majority voting to mask them. A vital element in this approach to fault tolerance is that replicated components must work on the same data: thus, if one node reads a sensor, it must distribute that reading to all the redundant copies of the application running in other nodes. Now a fault in that distribution mechanism could result in one node getting one value and another a different one (or no value at all). This would abrogate the requirement that all replicas obtain identical inputs, so we need to employ mechanisms to overcome this behavior.

The problem of distributing data consistently in the presence of faults is variously called *interactive consistency*, *consensus*, *atomic broadcast*, or *Byzantine agreement* [9, 12]. When a node transmits a message to several receivers, interactive consistency requires the following two properties to hold.

Agreement: All nonfaulty receivers obtain the same message (even if the transmitting node is faulty).

Validity: If the transmitter is nonfaulty, then nonfaulty receivers obtain the message actually sent.

Algorithms for achieving these requirements in the presence of arbitrary faults necessarily involve more than a single data exchange (basically, each receiver must compare the value it received against those received by others). It is provably impossible to achieve interactive consistency in the presence of a arbitrary faults unless there are at least $3a + 1$ FCUs, $2a + 1$ disjoint communication paths between them, and $a + 1$ levels (or "rounds") of communication. The number of FCUs and the number of disjoint paths required, but not the number of rounds, can be reduced by using digital signatures.

The problem might seem moot in architectures that employ a physical bus, since a bus surely cannot deliver values inconsistently (so the agreement property is achieved trivially). Unfortunately, it can—though it is likely to be a very rare event. The scenarios involving SOS faults presented earlier exemplify some possibilities.

Dealing properly with very rare events is one of the attributes that distinguishes a design that is fit for safety-critical systems from one that is not. It follows that either the application software must perform interactive consistency for itself (incurring the cost of n^2 messages to establish consistency across n nodes in the presence of a single arbitrary fault), or the bus architecture must do it.

The first choice is so unattractive that it vitiates the whole purpose of a fault-tolerant bus architecture. Most bus architectures therefore provide some type of interactively consistent message broadcast as a basic service. In addition, most architectures take steps to reduce the incidence of asymmetric transmissions (i.e., those that appear as one value to some receivers, and as different values, or the absence of values, to others). As noted, SOS faults are among the most plausible sources of asymmetric transmissions. SOS faults that cause asymmetric transmissions can arise in either the value or time domains (e.g., intermediate voltages, or weak edges, respectively). In those architectures that employ a bus guardian in a central hub or "in series" with each interface, the bus guardians are a possible point of intervention for the control of SOS faults: a suitable guardian can reshape, in both value and time domains, the signal sent to it by the controller. Of course, the guardian could be faulty and may make matters worse—so this approach makes sense only when there are independent guardians on each of two (or more) replicated interconnects. Observe that for credible signal reshaping, the guardian must have a power supply that is independent of that of the controller (faults in power supply are the most likely cause of intermediate voltages and weak edges).

Interactively consistent message broadcast provides the foundation for fault tolerance based on exact agreement. There are several ways to use this foundation. One arrangement, confusingly called the *state machine* approach [19], is based on majority voting: application replicas run on a number of different nodes, exchange their output values, and deliver a majority vote to the actuators.

Another arrangement is based on self-checking (either by individuals or pairs) so that faults result in fail-silence. This will be detected by other nodes, and some backup application running in those other nodes can take over. The architecture can assist this master/shadow arrangement by providing services that support the rollover from one node to another. One such service automatically substitutes a backup node for a failed master (both the master and the backup occupy the same slot in the schedule, but the backup is inhibited from transmitting unless the master is failed). A variant has both master and backup operating in different slots, but the backup inhibits itself unless it is informed that the master has failed. A further variation, called *compensation*, applies when different nodes have access to different actuators: none is a direct backup to any other, but each changes its operation when informed that others have failed (an example is car braking: separate nodes controlling the braking force at each wheel will redistribute the force when informed that one of their number has failed).

The variations on master/shadow described above all depend on a "failure notification," or equivalently a "membership" service. The crucial requirement on such a service is that it must produce *consistent* knowledge: that is, if one nonfaulty node thinks that a particular node has failed, then all other nonfaulty nodes must hold the same opinion—otherwise, the system will lose coordination, with potentially catastrophic results (e.g., if the nodes controlling braking at different wheels make different adjustments to their braking force based on different assessments of which others have failed). Notice that this must also apply to a node's knowledge of its *own* status: a naïve view might assume that a node that is receiving messages and seeing no problems in its own operation should assume it is in the membership. But if this node is unable to transmit, all other nodes will have removed it from their memberships and will be making suitable compensation on the assumption that this node has entered its "blackout" mode (and is, for example, applying no force to its brake). It could be catastrophic if this node does not adopt the consensus view and continues operation (e.g., applying force to its brake) based on its local assessment of its own health.

A membership service operates as follows. Each node maintains a private *membership* list, which is intended to comprise all and only the nonfaulty nodes. Since it can take a while to diagnose a faulty node, we have to allow the common membership to contain at most one faulty node. Thus, a membership service must satisfy the following two requirements.

Agreement: The membership lists of all nonfaulty nodes are the same.
Validity: The membership lists of all nonfaulty nodes contain all nonfaulty nodes and at most one faulty node.

These requirements can be achieved only under benign fault hypotheses (it is provably impossible to diagnose an arbitrary-faulty node with certainty). When unable to maintain accurate membership, the best recourse is to maintain agreement, but sacrifice validity (nonfaulty nodes that are not in the membership can then attempt to rejoin). This weakened requirement is called "clique avoidance" [2].

Note that it is quite simple to achieve consistent membership on top of an interactively consistent message service: each node broadcasts its own membership list to every other node, and each node runs a deterministic resolution algorithm on the (identical, by interactive consistency) lists received. Conversely, a membership and clique-

avoidance service can assist the construction of an interactively consistent message service: simply exclude from the membership any node that receives a message different than the majority (TTA does this).

5 Practical Implementations

Here, we provide sketches of four bus architectures that provide concrete solutions to the requirements and design challenges outlined in the previous sections. More details are available in a companion report to this paper [17]. All four buses support the time-triggered model of computation, employ fault-tolerant distributed clock synchronization, and use bus guardians or some equivalent mechanism to protect against babbling idiot failure modes. They differ in their fault hypotheses, mechanisms employed, services provided, and in their assurance, performance, and cost.

SAFEbus. Honeywell developed SAFEbus™ (the principal designers are Kevin Driscoll and Ken Hoyme [7]) to serve as the core of the Boeing 777 Airplane Information Management System (AIMS) [22], which supports several critical functions, such as cockpit displays and airplane data gateways. The bus has been standardized as ARINC 659 [1] and variations on Honeywell's implementation are being used or considered for other avionics and space applications. It uses a bus interconnect similar to that shown in Figure 1; the interfaces (they are called Bus Interface Units, or BIUs) are duplicated, and the interconnect bus is quad-redundant. Most of the functionality of SAFEbus is implemented in the BIUs, which perform clock synchronization and message scheduling and transmission functions. Each BIU of a pair is a separate FCU and acts as its partner's bus guardian by controlling its access to the interconnect.

Each BIU of a pair drives a different pair of interconnect buses but is able to read all four; the interconnect buses themselves each comprise two data lines and one clock line and operate at 30MHz. The bus lines and their drivers have the electrical characteristics of OR-gates (i.e., if several different BIUs drive the same line at the same time, the resulting signal is the OR of the separate inputs). Some of the protocols exploit this property; in particular, clock synchronization is achieved using an event-based algorithm.

The paired BIUs at sender and receiver, and the quad-redundant buses, provide sufficient redundancy for SAFEbus to provide interactively consistent message broadcasts (in the Honeywell implementation) using an approach similar to that described by Davies and Wakerly [4] (this remarkably prescient paper anticipated many of the issues and solutions in Byzantine fault tolerance by several years). It also supports application-level fault tolerance (based on self-checking pairs) by providing automatic rapid rollover from masters to shadows.

Its fault hypothesis includes arbitrary faults, faults in several nodes (but only one per node), and a high rate of fault arrivals. It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed its fault hypothesis. It tolerates spatial proximity faults in the AIMS application by duplicating the entire system. SAFEbus is certified for use in passenger aircraft and has extensive field experience in the Boeing 777. The Honeywell implementation is supported by an in-house tool chain.

SAFEbus is the most mature of the four buses considered, and makes the fewest compromises. But because each of its major components is paired (and its bus requires separate lines for clock and data), it is the most expensive of those available for commercial use (typically, a few hundred dollars per node).

TTA. The Time Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [8]. Commercial development of the architecture is undertaken by TTTech and it is being deployed for safety-critical applications in cars by Audi and Volkswagen, and for flight-critical functions in aircraft and aircraft engines by Honeywell.

Current implementations of TTA use a bus interconnect similar to that shown in Figure 1. The interfaces (they are called *controllers*) implement the TTP/C protocol [24] that is at the heart of TTA, providing clock synchronization, and message sequencing and transmission functions. The interconnect bus is duplicated and each controller drives both of them through partially independent bus guardians. TTA uses an averaging clock synchronization algorithm based on that of Lundelius and Lynch [25]. This algorithm is implemented in the controllers, but requires too many resources to be replicated in the bus guardians. The guardians, which have independent clocks, therefore rely on their controllers for a "start of frame" signal. This compromises their independence somewhat (they also share the power supply and some other resources with their controllers), so forthcoming implementations of TTA use a star interconnect similar to that shown in Figure 2. Here, the guardian functionality is implemented in the central hub which is fully independent of the controllers: the hubs and controllers comprise separate FCUs in this implementation. Hubs are duplicated for fault tolerance and located apart to withstand spatial proximity faults. They also perform signal reshaping to reduce the incidence of SOS faults.

TTA employs algorithms for group membership and clique avoidance [2]; these enable its clock synchronization algorithm to tolerate multiple faults (by reconfiguring to exclude faulty members) and combine with its use of checksums (which can be considered as digital signatures) to provide a form of interactively consistent message broadcasts. The membership service supports application-level fault tolerance based on master-backup or compensation. Proposed extensions provide state machine replication in a manner that is transparent to applications.

The fault hypothesis of TTA includes arbitrary faults, and faults in several nodes (but only one per node), provided these arrive at least two rounds apart (this allows the membership algorithm to exclude the faulty node). It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed this hypothesis.

The prototype implementations of TTA have been subjected to extensive testing and fault injections, and deployed in experimental vehicles. Several of its algorithms have been formally verified [13, 14], and aircraft applications under development are planned to lead to FAA certification. It is supported by an extensive tool suite that interfaces to standard CAD environments (e.g., Matlab/Simulink and Beacon). Current implementations provide 25 Mbit/s data rates; research projects are designing implementations for gigabit rates. TTA controllers and the star hub (which is basically a modified controller) are quite simple and cheap to produce in volume.

Of the architectures considered here, TTA is unique in being used for both automobile applications, where volume manufacture leads to very low prices, and aircraft, where a mature tradition of design and certification for flight-critical electronics provides strong scrutiny of arguments for safety.

SPIDER. A Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) is being developed by Paul Miner and colleagues at the NASA Langley Research Center as a research platform to explore recovery strategies for radiation-induced (HIRF/EMI) faults, and to serve as a case study to exercise the recent design assurance guidelines for airborne electronic hardware (DO-254) [15].

SPIDER uses a star configuration similar to that shown in Figure 2, in which the interfaces (called BIUs) may be located either with their hosts or in the centralized hub, which also contains active elements called Redundancy Management Units, or RMUs.

Clock synchronization and other services of SPIDER are achieved by novel distributed algorithms executed among the BIUs and RMUs [11]. The services provided include interactively consistent message broadcasts, and identification of failed nodes (from which a membership service can easily be synthesized). SPIDER's fault hypothesis uses a hybrid fault model, which includes arbitrary faults, and allows some combinations of multiple faults. Its algorithms are novel and highly efficient and are being formally verified.

SPIDER is an interesting design that uses a different topology and a different class of algorithms from the other buses considered here. However, it is a research project whose design and implementation are still in progress and so it cannot be compared directly with the commercial products.

FlexRay. A consortium including BMW, DaimlerChrysler, Motorola, and Philips, is developing FlexRay for powertrain and chassis control in cars. It differs from the other buses considered here in that its operation is divided between time-triggered and event-triggered activities. Published descriptions of the FlexRay protocols and implementation are sketchy at present [3] (see also the Web site `www.flexray-group.com`).

FlexRay can use either an "active" star interconnect similar to that shown in Figure 2, or a "passive" bus similar to that shown in Figure 1. In both cases, duplication of the interconnect is optional. The star configuration of FlexRay (and also that of TTA) can also be deployed in distributed configurations where subsystems are connected by hub-to-hub links. Each FlexRay interface (it is called a communication controller) drives the lines to its interconnects through separate bus guardians located with the interface. (This means that with two buses, each node has three clocks: one for the controller and one for each of the two guardians; this differs from the bus configuration of TTA where there is one clock for the controller and both guardians share a second clock.) Like the bus configuration of TTA, the guardians of FlexRay are not fully independent of their controllers.

FlexRay aims to be more flexible than the other buses considered here, and this seems to be reflected in the choice of its name. As noted, one manifestation of this flexibility is its combination of time- and event-triggered operation. FlexRay partitions each time cycle into a "static" time-triggered portion, and a "dynamic" event-triggered portion. The division between the two portions is set at design time and loaded into

the controllers and bus guardians. Communication during the event-driven portion of the cycle uses the Byteflight protocol. Unlike SAFEbus and TTA, FlexRay does not install the full schedule for the time-triggered portion in each controller. Instead, this portion of the cycle is divided into a number of slots of fixed size and each controller and its bus guardians are informed only of those slots allocated to their transmissions (nodes requiring greater bandwidth are assigned more slots than those that require less). Controllers learn the full schedule only when the bus starts up. Each node includes its identity in the messages that it sends; during startup, nodes use these identifiers to label their input buffers as the schedule reveals itself (e.g., if the messages that arrive in slots 1 and 7 carry identifier 3, then all nodes will thereafter deliver the contents of buffers 1 and 7 to the task that deals with input from node 3). There appears to be a vulnerability here: a faulty node could masquerade as another (i.e., send a message with the wrong identifier) during startup and thereby violate partitioning for the remainder of the mission. It is not clear how this fault mode is countered.

Like TTA, FlexRay uses the Lundelius-Welch clock synchronization algorithm but, unlike TTA, it does not use a membership algorithm to exclude faulty nodes. FlexRay provides no services to its applications beyond best-efforts message delivery; in particular, it does not provide interactively consistent message broadcasts. This means that all mechanisms for fault-tolerant applications must be provided by the applications programs themselves. Published descriptions of FlexRay do not specify its fault hypothesis, and it appears to have no mechanisms to counter certain fault modes (e.g., SOS faults or other sources of asymmetric broadcasts, and masquerading on startup). A never-give-up strategy has not been described, nor have systematic or formal approaches to assurance and certification.

FlexRay is interesting because of its mixture of time- and event-triggered operation, and potentially important because of the industrial clout of its developers. Currently, it is the slowest of the commercial buses, with a claimed data rate of no more than 10 Mbit/s.

6 Summary and Conclusion

A safety-critical bus architecture provides certain properties and services that assist in construction of safety-critical systems. As with any system framework or middleware package, these buses offer a tradeoff to system developers: they provide a coherent collection of services, with strong properties and highly assured implementations, but developers must sacrifice some design freedom to gain the full benefit of these services. For example, all these buses use a time-triggered model of computation, and system developers must build their applications within that framework. In return, the buses are able to guarantee strong partitioning: faults in individual components or applications ("functions" in avionics terms) cannot propagate to others, nor can they bring down the entire bus (within the constraints of its fault hypothesis).

Partitioning is the minimum requirement, however. It ensures that one failed function will not drag down others, but in many safety-critical systems the failure of even a single function can be catastrophic, so the individual functions must themselves be made fault tolerant. Accordingly, most of the buses provide mechanisms to assist the

development of fault-tolerant applications. The key requirement here is interactively consistent message transfer: this ensures that all masters and shadows (or masters and monitors), or all members of a voting pool, maintain consistent state. Three of the buses considered here provide this basic service; some of them do so in association with other services, such master/shadow rollover or group membership, that can be provided with much increased efficiency and much reduced latency when implemented at a low level. FlexRay, alone, provides none of these services. In their absence, all mechanisms for fault-tolerance must be implemented in applications programs. Thus, application programmers, who may have little experience in the subtleties of fault-tolerant systems, become responsible for the design, implementation, and assurance of very delicate mechanisms with no support from the underlying bus architecture. Not only does this increase the cost and difficulty of making sure that things are done right, it also increases their computational cost and latency. For example, in the absence of an interactively consistent message service provided by the architecture, applications programs must explicitly transmit the multiple rounds of cross-comparisons that are needed to implement this service at a higher level, thereby substantially increasing the message load. Such a cost will invite inexperienced developers to seek less expensive ways to achieve fault tolerance—in probable ignorance of the impossibility results in the theoretical literature, and the history of intractable "Heisenbugs" (rare, unrepeatable, failures) encountered by practitioners who pushed for 10^{-9} with inadequate foundations.

It is unlikely that any single bus architecture will satisfy all needs and markets, so it is possible that FlexRay's lack of application-level fault-tolerant services will find favor in some areas. It is also to be expected that new or modified architectures will emerge to satisfy new markets and requirements. (For example, it is proposed that TTA could match FlexRay's ability to support event-triggered as well as time-triggered communications by allocating certain time slots to a simulation of CAN; the simulation is actually faster than a real CAN bus, while retaining all the safety attributes of TTA.) I hope that the description provided here will help potential users to evaluate existing architectures against their own needs, and that it will help designers of new architectures to learn from and build on the design choices made by their predecessors.

References

1. *ARINC Specification 659: Backplane Data Bus*. Aeronautical Radio, Inc, Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.
2. Günther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *19th Symposium on Reliable Distributed Systems*, Nuremberg, Germany, October 2000.
3. Joef Berwanger et al. FlexRay–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Society of Automotive Engineers, Detroit, MI, April 2001. Paper number 2001-01-0676.
4. Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978.
5. *System Design and Analysis*. Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

6. Günter Heiner and Thomas Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Fault Tolerant Computing Symposium 28*, pages 402–407, IEEE Computer Society, Munich, Germany, June 1998.
7. Kenneth Hoyme and Kevin Driscoll. SAFEbus™. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 68–73, Seattle, WA, October 1992.
8. Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.
9. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
10. Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.
11. Paul S. Miner. Analysis of the SPIDER fault-tolerance protocols. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, NASA Langley Research Center, Hampton, VA, June 2000. Slides available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Presentations/lfm2000-spider/`.
12. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
13. Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, eds., *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, Oct. 2000.
14. Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, eds., *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, Jan. 1999.
15. *DO254: Design Assurance Guidelines for Airborne Electronic Hardware*. Requirements and Technical Concepts for Aviation, Washington, DC, April 2000.
16. John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Association for Computing Machinery, Los Angeles, CA, August 1994. Also available as NASA Contractor Report 198289.
17. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 2001. Available at `http://www.csl.sri.com/~rushby/papers/buscompare.pdf`.
18. Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1987.
19. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
20. Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
21. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
22. William Sweet and Dave Dooling. Boeing's seventh wonder. *IEEE Spectrum*, 32(10):20–23, October 1995.
23. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, IEEE Computer Society, Columbus, OH, October 1988.
24. *Specification of the TTP/C Protocol*. Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria, July 1999.
25. J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

On the Composition of Real-Time Schedulers *

Weirong Wang Aloysius K. Mok

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
{weirongw, mok}@cs.utexas.edu

Abstract. A complex real-time embedded system may consist of multiple application components each of which has its own timeliness requirements and is scheduled by component-specific schedulers. At run-time, the schedules of the components are integrated to produce a system-level schedule of jobs to be executed. We formalize the notions of schedule composition, task group composition and component composition. Two algorithms for performing composition are proposed. The first one is an extended Earliest Deadline First algorithm which can be used as a composability test for schedules. The second algorithm, the Harmonic Component Composition algorithm (HCC) provides an online admission test for components. HCC applies a rate monotonic classification of workloads and is a hard real-time solution because responsive supply of a shared resource is guaranteed for in-budget workloads. HCC is also efficient in terms of composability and requires low computation cost for both admission control and dispatch of resources.

1 Introduction

The integration of components in complex real-time and embedded systems has become an important topic of study in recent years. Such a system may be made up of independent application (functional) components each of which consists of a set of tasks with its own specific timeliness requirements. The timeliness requirements of the task group of a component is guaranteed by a scheduling policy specific to the component, and thus the scheduler of a complex embedded system may be composed of multiple schedulers. If these components share some common resource such as the CPU, then the schedules of the individual components are interleaved in some way. In extant work, a number of researchers have proposed algorithms to integrate real-time schedulers such that the timeliness requirements of all the application task groups can be simultaneously met. The

* This work is supported in part by a grant from the US Office of Naval Research under grant number N00014-99-1-0402 and N00014-98-1-0704, and by a research contract from SRI International under a grant from the NEST program of DARPA

most relevant work in this area includes work in "open systems" and "hierarchical schedulers" which we can only briefly review here. Deng and Liu proposed the open system environment, where application components may be admitted online and the scheduling of the component schedulers is performed by a kernel scheduler [2]. Mok and Feng exploited the idea of temporal partitioning [5], by which individual applications and schedulers work as if each one of them owns a dedicated "real-time virtual resource". Regehr and Stankovic investigated hierarchical schedulers [7]. Fohler addressed the issue of how to dynamically schedule event-triggered tasks together with an offline-produced schedule for time-triggered computation [3]. In [9] by Wang and Mok, two popular schedulers: the cyclic executive and fixed-priority schedulers form a hybrid scheduling system to accommodate a combination of periodic and sporadic tasks.

All of the works cited above address the issue of schedule/scheduler composition based on different assumptions. But what exactly are the conditions under which the composition of two components is correct? Intuitively, the minimum guarantee is that the composition preserves the timeliness of the tasks in all the task groups. But in the case an application scheduler may produce different schedules depending on the exact time instants at which scheduling decisions are made, must the composition of components also preserve the exact schedules that would be produced by the individual application schedulers if they were to run on dedicated CPUs? Such considerations may be important if an application programmer relies on the exact sequencing of jobs that is produced by the application scheduler and not only the semantics of the scheduler to guarantee the correct functioning of the application component. For example, an application programmer might manipulate the assignment of priorities such that a fixed priority scheduler produces a schedule that is the same as that produced by a cyclic executive for an application task group; this simulation of a cyclic executive by a fixed priority scheduler may create trouble if the fixed priority scheduler is later on composed with other schedulers and produces a different schedule which does not preserve the task ordering in the simulated cyclic executive. Hence, we need to pay attention to semantic issues in scheduler composition.

In this paper, we propose to formalize the notions of composition on three levels: schedule composition, task group composition and component composition. Based on the formalization, we consider the questions of whether two schedules are composable, and how components may be efficiently composed. Our formalization takes into account the execution order dependencies (explicit or implicit) between tasks in the same component. For example, in cyclic executive schedulers, a deterministic order is imposed on the execution of tasks so as to satisfy precedence, mutual exclusion and other relations. As is common practice to handle such dependencies, sophisticated search-based algorithms are used to produce the deterministic schedules offline, e.g., [8]. To integrate such components into a complex system, we consider composition with the view that: First, the correctness of composition should not depend on knowledge about how the component schedules are produced, i.e., compositionality is fundamentally a predicate on *schedules* and not *schedulers*. Second, the composition of schedules

should be *order preserving* with respect to its components, i.e., if job x is scheduled before job y in a component schedule, then job x is still scheduled before y in the integrated system schedule. Our notion of *schedule composition* is an interleaving of component schedules that allows preemptions between jobs from different components.

The contributions of this paper include: formal definitions of schedule composition, task group composition and component composition, an optimal schedule composition algorithm for static schedules and a harmonic component composition algorithm that has low computation cost and also provides a responsiveness guarantee. The rest of the paper is organized as follows. Section 2 defines basic concepts used in the rest of the paper. Section 3 addresses schedule composition. Section 4 defines and compares task group composition and component composition. Section 5 defines, illustrates and analyzes the Harmonic Component Composition approach. Section 6 compares HCC with related works. Section 7 concludes the paper by proposing future work.

2 Definitions

2.1 Task Models

Time is defined on the domain of non-negative real numbers, and the time interval between time b and time e is denoted by (b, e). We shall also refer to a time interval $(i, i+1)$ where i is a non-negative integer as a *time unit*.

A *resource* is an object to be allocated to tasks. It can be a CPU, a bus, or a packet switch, etc. In this paper, we shall consider the case of a single resource which can be shared by the tasks and components, and preemption is allowed. We assume that context switching takes zero time; this assumption can be removed in practice by adding the appropriate overhead to the task execution time.

A *job* is defined by a tuple of three attributes (c, r, d) each of which is a non-negative real number:

- c is the *execution time* of a job, which defines the amount of time that must be allocated to the job;
- r is the *ready time* or *arrival time* of the job which is the earliest time at which the job can be scheduled;
- d is the *deadline* of the job which is the latest time by which the job must be completed.

A *task* is an infinite sequence of jobs. Each task is identified by a unique ID i. A task is either periodic or sporadic.

The set of periodic tasks in a system is represented by T_p. A *periodic task* is denoted by $(i, (c, p, d))$, where i identifies the task, and tuple (c, p, d) defines the attributes of its jobs. The jth job of i is denoted by job (i, j).

Suppose X identifies an object and Y is one of the attributes of the object. we shall use the notation $X.Y$ to denote the attribute Y of X. For instance, if (i, j) identifies a job, then $(i, j).d$ denotes the deadline of job (i, j).

The attributes in the definition of a periodic task, c, p and d, are non-negative real numbers:

- c is the *execution time* of a task, which defines the amount of time that must be allocated to each job of the task;
- p is the *period* of the task;
- d is the *relative deadline* of the task, which is the maximal length of time by which a job must be completed after its arrival. We assume that for every periodic task, $c \leq d \leq p$.

If a periodic task i is defined by (c, p, d), job (i, j) is defined by $(c, j \cdot p, j \cdot p + d)$.

A *sporadic task* is denoted by a tuple $(i, (c, p, d))$, where i identifies the task, and (c, p, d) defines the attributes of its jobs, as follows: The jth job of sporadic task i is identified as job (i, j), $j \geq 0$. The arrival times of jobs of a sporadic task are not known *a priori* and are determined at run time by an *arrival function* A that maps each job of a sporadic task to its arrival time for the particular run:

$A :: T_s \times \mathbf{N} \rightarrow \mathbf{R}$, where $\mathbf{N}$ is the set of natural numbers and $\mathbf{R}$ is the set of real numbers.
$A(i, j) = t$ if the job (i, j) arrives at time t.
$A(i, j) = \perp$ if the job (i, j) never arrivals.

The attributes c and d of a sporadic task are defined the same as those of a periodic task. However, attribute p of a sporadic task represents the *minimal* interval between the arrival times of any two consecutive jobs. In terms of the function A, $A(i, (j + 1)) - A(i, j) \geq p$ if $A(i, (j + 1))$ is defined.

For a sporadic task $(i, (c, p, d))$, job (i, j) is defined as $(c, A(i, j), A(i, j) + d)$.

A task group TG consists of a set of tasks (either periodic or sporadic). We shall use STG to denote a set of task groups. The term component denotes a task group and its scheduler. Sometimes we call a task group an application task group to emphasize its association with a component which is one of many applications in the system.

2.2 Schedule

A resource supply function Sup defines the maximal time that can be supplied to a component from time 0 to time t. Time supply function must be monotonically non-decreasing. In other words, if $t \leq t'$, then $Sup(t) \leq Sup(t')$.

The function S maps each job to a set of time intervals:

$S :: TG \times \mathbf{N} \rightarrow \{(\mathbf{R}, \mathbf{R})\}$ where TG is a task group, and $\mathbf{N}$ and $\mathbf{R}$ represent the set of natural numbers and the set of real numbers respectively.
$S(i, j) = \{(b_{i,j,k}, e_{i,j,k}) | 0 \leq k < h\}$ where k and h are natural numbers.

S is a *schedule* of TG under supply function Sup if and only if all of the following conditions are satisfied:

- **Constraint 1**: For every job (i, j), every time interval assigned to it in the schedule must be assigned in a time interval allowed by the supply function, i.e., for all $(b, e) \in S(i, j)$, $Sup(e) - Sup(b) = e - b$.

- **Constraint 2**: The resource is allocated to at most one job at a time, i.e., time intervals do not overlap: For every $(b_{i,j,k}, e_{i,j,k}) \in S(i,j)$ and for every $(b_{i',j',k'}, e_{i',j',k'}) \in S(i',j')$, one of the following cases must be true:
  - $e_{i,j,k} \leq b_{i',j',k'}$; or
  - $e_{i',j',k'} \leq b_{i,j,k}$; or
  - $i = i'$, $j = j'$ and $k = k'$.
- **Constraint 3**: A job must be scheduled between its ready time and deadline: for every $(b,e) \in S(i,j)$,

$$(i,j).r \leq b < e \leq (i,j).d$$

- **Constraint 4**: For every job (i,j), the total length of all time intervals in $S(i,j)$ is sufficient for executing the job, i.e.,

$$\sum_{(b,e) \in S(i,j)} (e - b) \geq (i,j).c$$

Given a time t, if there exists a time interval (b,e) in $S(i,j)$ such that $b \leq t < e$, then job (i,j) is *scheduled at* time t, and task i is *scheduled at* time t.

An algorithm Sch is a *scheduler* if and only if it produces a schedule S for T under A and Sup.

A component C of a system is defined by a tuple (TG, Sch) which specifies the task group to be scheduled and the task group's scheduler. A set of components will be written as SC.

3 Schedule Composition

Suppose S_h is a schedule of a component task group TG_h. We say that the schedule S integrating the component schedules in $\bigcup TG_h$ is a composed schedule of all component schedules $\{S_h | 0 \leq h \leq n-1\}$ if and only if there exists a function M which maps each scheduled time interval in S_h to a time window subject to the following conditions:

- For each time interval $(b,e) \in S_h(i,j)$, $M(h,(b,e)) = (b_h, e_h)$, and (b_h, e_h) is within the ready time and deadline of job (i,j);
- The time scheduled to job (i,j) by S between (b_h, e_h) is equal to $e - b$:

$$\sum_{(x,y) \in S(i,j) \text{ and } b_h \leq x \leq y \leq e_h} (y - x) = e - b$$

- $M(h,(b,e))$ is before $M(h,(b',e'))$ if and only if $(b,e) \in S_h(i,j)$ is before $(b',e') \in S_h(i',j')$.

The notion of schedule composition is illustrated in Figure 1 where the component schedule S_0 is interleaved with other component schedules into a composed schedule S. Notice that the time intervals occupied by S_0 can be mapped into S without changing the order of these time intervals.

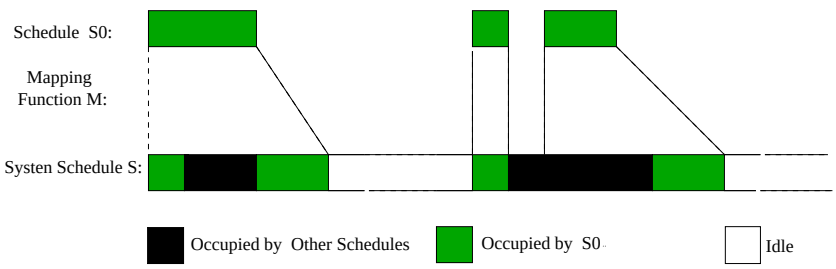

Fig. 1. Definition of Schedule Composition

To test whether a set of schedules can be integrated into a composed schedule, we now propose an extended Earliest Deadline First algorithm for schedule composition. From the definition of a schedule, the execution of a job (i, j) can be scheduled into a set of time intervals by a schedule S. We use the term $S(i, j)$ to denote the set of time intervals job (i, j) occupies. In the following, we shall refer to a time interval in $S(i, j)$ as a *job fragment* of the job (i, j). The schedule composition algorithm works as follows. A job fragment is created corresponding to the first time interval of the first job in each component schedule S_h that has not been integrated into S, and the job fragments from all schedules are scheduled together by EDF. After the job fragment, say for schedule S_h has completed, the job fragment is deleted and another job fragment is created corresponding to the next time interval in schedule S_h.

The schedule composition algorithm is defined below.

- Initially, all job fragments from all component schedules are unmarked.
- At any time t, *Ready* is a set that contains all the job fragments from all the component schedules that are ready to be composed. Initially, *Ready* is empty.
- At any time t, if there is no job fragment from component schedule S_h in *Ready*, construct one denoted as (h, c, r, d) by the following steps:
  - Let (b, e) be an unmarked time interval such that $(b, e) \in S_h(i, j)$ and for all unmarked time interval $(b', e') \in S_h(i', j')$, $b \leq b'$;
  - Define the execution time of the job fragment as the length of the scheduled time interval: $c := e - b$;
  - Define ready time of the job fragment as the ready time of the job scheduled at (b, e): $r := (i, j).r$;
  - Define deadline of the job fragment as the earliest deadline among all jobs scheduled after time b by S_h:

$$d := min(\{(i', j').d | (b', e') \in S_h(i', j') \text{ and } b \leq b'\})$$

  - Mark interval (b, e).
- Allocate the resource to the job fragment in *Ready* that is ready and has the earliest deadline.

- If the accumulated time allocated to job fragment is equal to the execution time of the job fragment, delete the job fragment from *Ready*.
- If t is equal to the deadline of a job fragment before the completion of the corresponding job in *Ready*, the schedule composition fails.

In the above, the time intervals within a component schedule S_h are transformed into job fragments and put into *Ready* one by one in their original order in S_h. At any time t, just one job fragment from S_h is in *Ready*. Therefore, the order of time intervals in a component schedule is preserved in the composed schedule.

The extended EDF is optimal in terms of composability. In other words, if a composed schedule exists for a given set of component schedules, then the extended EDF produces one.

Theorem 1 *The extended EDF is an optimal schedule composition algorithm.*

Proof: If the extended EDF for composition fails at time f, then let s be the latest time that following conditions are all true: for any S_h, there exists $(b, e) \in S_h(i, j)$, $(i, j).r \geq s$, all time intervals before b in S_h are composed into S no later than time s, and for all (b', e') composed between s and f, the corresponding job fragment has deadline no later than f. Then for any time t between (s, f), there is a $(b', e') \in S(i', j')$ and $b' \leq t \leq e'$. The aggregate length of time intervals from component schedules that must be integrated between (s, f) is larger than $f - s$, therefore no schedule composition exists. ■

Because of its optimality, the extended EDF is a composability test for any set of schedules. Although extend EDF is optimal, this approach, however, has a limitation: the input component schedules must be static. In other words, to generate system schedule at time t, the component schedules after time t need to be known. Otherwise, the deadline of the pseudo job in *Ready* cannot be decided optimally. Therefore, the extended EDF schedule composition approach cannot be applied optimally to dynamically produced schedules.

4 Task Group Composability and Component Composability

We say that a set of task groups STG=$\{TG_0, .., TG_{n-1}\}$ is *weakly* composable if and only if the following holds: Given any set of arrival functions $\{A_0, .., A_{n-1}\}$ for the task groups in STG, for any $0 \leq k \leq n-1$, there exists a schedule S_k for TG_k under A_k, and $SS = \{S_0, .., S_{n-1}\}$ is composable. Obviously, weak composability is equivalent to the schedulability of task group $\bigcup_{STG} TG_k$.

We say that a set of task groups STG is *strongly* composable if and only if the following holds: Given any schedule S_k of TG_k under any A_k, $SS = \{S_0, .., S_{n-1}\}$ is composable. The following is a simple example of strong composability.

Suppose there are two task groups. TG_0 consists of a periodic task $T_0 = (1, 5, 5)$, and TG_1 consists of a sporadic task $T_1 = (1, 5, 5)$. Then an arbitrary schedule S_0 for TG_0 and an arbitrary schedule S_1 of TG_1 can always be composed

into a schedule S by the extended EDF no matter what the arrival function is. Therefore, this set of task groups are strongly composable.

Not all *weakly* composable sets of task groups are strongly composable. Suppose we change the above example of strongly composable set of task groups by adding another periodic task $T_2 = (4, 10, 10)$ to task group TG_0. Two schedules can be produced for TG_0 by a fixed priority schedulers: S_0 and S_0'. In S_0, suppose we give a higher priority to T_0, and therefore for all j, $S_0(0, j) = (5 \cdot j, 5 \cdot j + 1)$, and $S_0(2, j) = (10 \cdot j + 1, 10 \cdot j + 5)$. For S_0', suppose we give higher priority to T_2, and therefore for any number j, $S_0'(0, 2j) = (10 \cdot j + 4, 10 \cdot j + 5)$, $S_0'(0, 2j + 1) = (10 \cdot j + 5, 10 \cdot j + 6)$; $S_0'(2, j) = (10 \cdot j, 10 \cdot j + 4)$.

S_0 is composable with any schedule S_1 of TG_1, but S_0' is not. In S_0', for any j, the deadline of job $(0, 2 \cdot j)$ is at $10 \cdot j + 5$, and yet it is scheduled after job $(2, j)$ whose deadline is at $10 \cdot j + 10$. Because of the order-preserving property of schedule composition, it follows that every time interval $(10 \cdot j,\ 10 \cdot j + 5)$ must be assigned to S_0'. Thus, if a job of T_1 arrives at time $10 \cdot j$, schedule composition becomes impossible.

We say that a set of supply functions $SSup$=$\{Sup_0, .., Sup_{n-1}\}$ is *consistent* if and only if the aggregate time supply of all functions between any time interval (b, e) is less than or equal to the length:

$$\sum (Sup_k(e) - Sup_k(b)) \leq e - b$$

Suppose $SC = \{(Sch_0, TG_0), .., (Sch_{n-1}, TG_{n-1})\}$ is a set of components. SC is composable if and only if given any set of arrival functions $SA = \{A_0, .., A_{n-1}\}$, there exists a set of consistent supply functions $SSup = \{Sup_0, .., Sup_{n-1}\}$ such that Sch_k produces schedule S_k of TG_k under arrival function A_k and supply function Sup_k, and $SS = \{S_0, .., S_{n-1}\}$ is composable.

Component composability lies between weak composability and strong composability of task groups in the following sense. A component has its own scheduler which may produce for a given arrival function, a schedule among a number of valid schedules under the arrival function. Therefore, given a set of components, if the corresponding set of task groups of these components are strongly composable, then the components are composable; if the task groups are not even weakly composable, the components are not composable. However, when the task groups are weakly but not strongly composable, component composability depends on the specifics of component schedulers.

To illustrate these concepts, we compare weak task group composability, strong task group composability and component composability in the following example which is depicted in Figure 2. Suppose there are two components $C_0 = (TG_0, Sch_0)$ and $C_1 = (TG_1, Sch_1)$. For any valid arrival function A for each of the task groups, there exists in general a set of schedules that may correspond to the execution of the task group under the arrival function set. In Figure 2, the circle marked as $SS_{0,0}$ represents the set for all possible schedules of TG_0 under A_0; and $SS_{0,1}$, $SS_{1,0}$, $SS_{1,1}$ are defined similarly. If TG_0 and TG_1 are strongly composable, then randomly pick a schedule S_0 from $SS_{0,x}$ and a schedule S_1 from $SS_{1,y}$ where x and y are variable and S_0 and S_1 are composable. If TG_0

and TG_1 are weakly composable, then for any x and y, there exists a schedule S_0 from $SS_{0,x}$ and there exists a schedule S_1 from $SS_{1,y}$ such that S_0 and S_1 are composable. The small circle marked as $SS_{0,0,s}$ is the set of all schedules that can be produced by the scheduler Sch_0 under A_0. Each point in $SS_{0,0,s}$ corresponds to one schedule, and one or multiple supply functions upon which Sch_0 produces $SS_{0,0,s}$. Circle $SS_{0,1,s}$, $SS_{1,0,s}$, $SS_{1,1,s}$ are defined similarly. If components C_0 and C_1 are composable, then for any pair of x and y, there exists a schedule S_0 in $SS_{0,x,s}$, and a schedule S_1 in $SS_{1,y,s}$, S_0 and S_1 are composable, and there exists a supply function Sup_0 corresponding to S_0 and a supply function Sup_1 corresponding to S_1, and Sup_0 and Sup_1 are consistent.

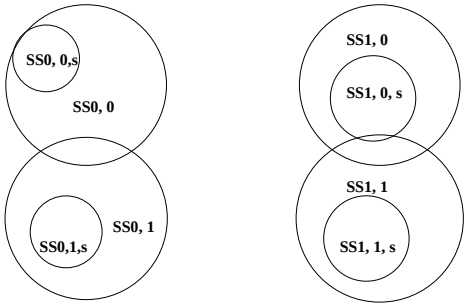

Fig. 2. Composability

In many scheduler composition paradigms, the resource supply functions can be determined only online for components that have unpredictable arrivals of jobs. Therefore it is often hard to define resource supply function *a priori*. However, we can introduce the notion of contracts to express the requirements imposed on the supply function by a component, as the interface between a component and the composition coordinator. In the next section, we shall discuss Harmonic Component Composition which makes use of explicit supply function contracts.

5 Harmonic Component Composition

We consider the tradeoff between composability and the simplicity in the design of the system-level scheduler to be a significant challenge in component composition. As an extreme case in pursuing simplicity, a coordinator may allocate resources among components based on a few coarse-grain parameters of each component, such as the worst case response time and bandwidth requirement. This type of solutions often does not achieve composability, i.e., admission of new components may be disallowed even when the aggregate resource utilization is low because of previous overly conservative capacity commitments. At the

opposite extreme, the coordinator may depend on details about the components to perform complex analysis and may take on too many obligations from individual components, such that the system performance may eventually be degraded. We now propose a solution to meet the challenge by introducing class-based workloads. We call this approach Harmonic Component Composition (HCC).

5.1 Coordinator Algorithm

The system designer will select a constant K as the number of resource classes. A class k ($k \in [0, K)$) is defined by a class period $P_k = m^k$, where m is a designer-selected constant. We require a rate monotonic relation between the periods of classes: For any $0 \leq l \leq k \leq K-1$, $\frac{P_k}{P_l} = m^{l-k}$. Lower class has larger class number and longer class period.

When a component C is ready to run, it generates a supply contract and sends it to the coordinator. The supply contract is a list of workload defined as (k, l, w), where $k \leq l$. The workload permits that up to w time units of resource supply can be on demand within any time interval of length m^l; and once a demand occurs, it must be met within m^k time units. Upon receiving a supply contract, the coordinator will admit a component if and only if it can satisfy the contract without compromising the contracts with previously admitted components.

When a demand is proposed to class k, it will be served within m^k time. To keep this guarantee, HCC maintains a straightforward invariant to make sure that supply needed online for class k or higher in any time interval with length m^k is less than or equal to m^k. To accomplish this, the aggregate workload admitted to class k or higher is constrained as if there is a *conceptual resource* associated with class k which is consumed by admitting any workload with class k or higher. Suppose that R_k represents the conceptual resource of class k. R_k is initiated as P_k. A workload (k, l, w) requires no conceptual resource from the classes higher than k, but requires that from every class lower than or equal to k. The value of the conceptual resource requirement of a workload (k, l, w) on class i is derived from the worst case occupation in a time interval of length P_i by the workload.

If a component C_h is admitted, the coordinator establishes a server identified with (h, k, l) for each workload (k, l, w) in the contract. The component to which the server belongs is identified by h, the class of the server is k, and (k, l) defines a subclass. All servers of class i are in a list L_i. The server is defined with a budget limit w and replenishment period of m^l. A server have four registers, *load*, *carry*, *budget* and *replenish*.

Initialization:

(1) **foreach** $0 \leq k \leq K-1$
(2) $\quad R_k := P_k$
(3) $\quad L_k$ is set as an empty list

Contract Admission:

(1) Upon component C_h proposes a contract V_h, which is a list of (k, l, w)
(2) **foreach** $0 \leq i \leq K-1$
(3) $\quad R'_i := R_i$
(4) **foreach** $(k, l, w) \in V_h$
(5) $\quad$ **foreach** $k \leq i \leq l$
(6) $\quad\quad R'_i := R'_i - w$
(7) $\quad$ **foreach** $l+1 \leq i \leq K-1$
(8) $\quad\quad R'_i := R'_i - w \cdot (m^{i-l})$
(9) $\quad$ **if** $\exists R'_i < 0$
(10) $\quad\quad$ reject component C_h and terminate this run of contract admission;
(11) **foreach** $i \in [0, K-1]$
(12) $\quad R_i := R'_i$
(13) **foreach** $(k, l, w) \in V_h$
(14) $\quad$ construct server (h, k, l) and add to the end of L_k, with the following initial values:
(15) $\quad$ $budget = w$, $loaded = carry = 0$, $replenish$ as empty queue.

Referring to the algorithm specification above, a component C_h may *load* a server (h, k, l) by adding a value to its register *load* when the component C_h demands usage on the resource. If the value of the *load* register is positive, the server is *loaded*. If a loaded server has budget($budget > 0$), then the budget is consumed on the load and all or part of the loaded value becomes *carried* ($carry > 0$). At the start of a time unit $(t, t+1)$ (which means t is a non-negative integer), if class k is the highest class with a carried server, then the first carried server in L_k supplies resource in the time unit $(t, t+1)$.

The existing budget of a server is held in *budget*. When *load* and *budget* are both positive and $v = min(load, budget)$, both of them are reduced by v and *carry* is increased by v. Consumed budget will be replenished after m^l units of time. The queue *replenish* records the scheduled replenishments in the future.

Online Execution:

(1) Upon the start of time unit $(t, t+1)$:
(2) **foreach** server (h, k, l)
(3) Replenish budget:
(4) **if** the head of queue in *replenish* is (t, val)
(5) $budget := budget + val$
(6) dequeue (t, val) from *repenish*
(7) Carry work load:
(8) **if** $load > 0$ and $budget > 0$
(9) $v := min(load, budget)$
(10) $carry := carry + v$
(11) $budget := budget - v$
(12) $load := load - v$
(13) enqueue $(v, t + m^l)$ to *replenish*
(14) Supply Resource:
(15) Select server (h, k, l), such that k is the highest class with at least one carried server, and (h, k, l) is the first carried server in L_k.
(16) $carry := carry - 1$
(17) Supply resource to component C_h in time unit $(t, t+1)$

When a component terminates, the coordinator reclaims the conceptual resources from the component.

Component termination:

(1) Upon the termination of component C_h
(2) **foreach** $(k, l, w) \in V_h$
(3) delete server (h, k, l) from L_k
(4) **foreach** $k \leq i \leq l$
(5) $R_i := R_i + w$
(6) **foreach** $l + 1 \leq i \leq K - 1$
(7) $R_i := R_i + w \cdot (m^{i-l})$

5.2 Component Algorithm

In the HCC approach, a component generates a supply contract, and if admitted, it may demand supply from its servers. Different algorithms may be applied for different components in a composition. We describe one solution here as an example.

Assume that there is a component C_h, and its component scheduler is EDF. A task (c, p, d) is categorized to subclass $(\lfloor log_m d \rfloor, \lfloor log_m p \rfloor)$, and its execution time is added to the weight w of the workload with that subclass.

Supply Contract Generation:

(1) **foreach** (k, l) such that $0 \leq k \leq l \leq K - 1$
(2) $\quad w_{k,l} := 0$
(3) **foreach** $i \in T_h$
(4) $\quad k := \lfloor \log_m i.d \rfloor$
(5) $\quad l := \lfloor \log_m i.p \rfloor$
(6) $\quad w_{k,l} := w_{k,l} + i.c$
(7) **foreach** $w_{k,l} \neq 0$
(8) $\quad$ add workload $(k, l, w_{k,l})$ into contract V_h

At run time, upon the arrival of a job (i, j), a demand for resource supply is added to the server corresponding to task i at the start of the next time unit.

Online execution:

(1) Initialization:
(2) $\quad$ **foreach** (k, l)
(3) $\quad\quad w_{k,l} := 0;$
(4) Upon the arrival of job (i, j),
(5) $\quad k := \lfloor \log_m i.d \rfloor$
(6) $\quad l := \lfloor \log_m i.p \rfloor$
(7) $\quad w_{k,l} := w_{k,l} + i.c$
(8) Upon the start of time unit $(t, t + 1)$
(9) $\quad$ **foreach** server (k, l) such that $w_{k,l} > 0$
(10) $\quad\quad load := load + w_{k,l};$
(11) $\quad\quad w_{k,l} := 0;$

5.3 Example

Having described how HCC works, we illustrate the HCC approach by an example below.

In this example, we design a system with four components with the following specifications.

- Component C_0 consists of one task for emergency action and 2 periodic routine tasks. The emergency action takes little execution time and rarely happens, but when a malfunction occurs, the action must be performed immediately. We abstract this action by a sporadic task $T_0 = (1, \infty, 1)$, which means that the execution time and relative deadline are both 1, and the minimum interval between consecutive arrivals are infinite. The periodic routine tasks are given by $T_1 = (1, 80, 8)$, $T_2 = (1, 100, 10)$.

- Component C_1 is a group of periodic routine tasks defined as follows: $T_3 = (1, 3, 3)$, $T_4 = (1, 10, 10)$.
- Component C_2 is a bandwidth-intensive application, which needs 25 percent of the resource. It can be modeled as $T_5 = (16, 64, 64)$.
- Component C_3 has one periodic task $T_6 = (3, 30, 30)$.

The value of m and K are arbitrarily selected as 2 and 6 by the system designer, based on estimations of the potential workloads. Let us apply the contract generation as defined in this paper. Four contracts will be produced as follows. Recall that workload is defined as (k, l, w).

- $V_0 = \{(0, 6, 1), (3, 6, 2)\}$, where T_0 is mapped to workload $(0, 6, 1)$, T_1 and T_2 are mapped to $(3, 6, 2)$.
- $V_1 = \{(1, 1, 1), (3, 3, 1)\}$, where T_3 is mapped to $(1, 1, 1)$, and T_4 is mapped to $(3, 3, 1)$.
- $V_2 = \{(6, 6, 16)\}$.
- $V_3 = \{(4, 4, 3)\}$.

Suppose that all components become ready at time 0, and the admission decisions are made according to their index order. For all $0 \leq k \leq 6$, R_k remains non-negative when C_0, C_1, C_2 are admitted. However, during the admission of C_3, $R'_6 < 0$, therefore C_3 is not admitted. Table 1 shows the change of R_k during admission procedure, and Table 2 shows the established servers on all classes after that.

Table 1. Component Admission

		Component 0		**Component 1**		**Component 2**		**Component 3**
	initial	(0, 6, 1)	(3, 6, 2)	(1, 1, 1)	(3, 3, 1)	(6, 6, 16)		(4, 4, 3)
R_0	1	0	0	0	0	0	R'_0	0
R_1	2	1	1	0	0	0	R'_1	0
R_2	4	3	3	1	1	1	R'_2	1
R_3	8	7	5	1	0	0	R'_3	0
R_4	16	15	13	5	3	3	R'_4	0
R_5	32	31	29	13	9	9	R'_5	3
R_6	64	63	61	29	21	5	R'_6	-7

Assume that the first job of T_0 arrives at time 4 and the online executions of all components are defined as in this paper. We now show a step by step execution from time 0 to time 4.

At time 0, the budget registers of all servers have been initialized according to their weights, and the components add their current demands to the corresponding load registers, as shown in Table 3. Coordinator moves the in-budget loads into register *carry*, and the consumed budget are recorded for replenishments in the future. The carried value of server $(1, 1, 1)$ becomes 1. Server $(0, 0, 6)$ is not

Table 2. Servers on All Classes

L_0	$\{(0,0,6)\}$
L_1	$\{(1,1,1)\}$
L_2	
L_3	$\{(0,3,6),(1,3,3)\}$
L_4	
L_5	
L_6	$\{(2,6,6)\}$

carried, therefore server $(1,1,1)$ is selected to supply time between time $(0,1)$. Its *carry* is then decremented back to 0. Table 4 shows the register image after the execution of the coordinator.

Table 3. Register Image Right After Component Loading At Time 0

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	1	1	0	
(0, 3, 6)	2	2	0	
(1, 3, 3)	1	1	0	
(2, 6, 6)	16	16	0	

Table 4. Register Image Right After Coordinator Execution At Time 0

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	0	0	0	$\{(1,2)\}$
(0, 3, 6)	0	0	2	$\{(2,64)\}$
(1, 3, 3)	0	0	1	$\{(1,8)\}$
(2, 6, 6)	0	0	16	$\{(16,64)\}$

Between time $(0,1)$, no load is added from any component. At time 1, server $(0,3,6)$ is selected to supply between $(1,2)$ so its *carry* is decremented, as shown in Table 5.

At time 2, server (1, 1, 1) replenishes its budget, and server (0, 3, 6) is selected as supplier and so its value of *carry* is decremented, as shown in Table 6.

At time 3, the second job of T_3 is ready, so C_1 loads server $(1,1,1)$ by 1, as shown in Table 7. On the coordinator side, budget is available for server $(1,1,1)$, therefore budget is consumed for the load and *carry* is incremented by

Table 5. Register Image Right After Coordinator Execution At Time 1

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	0	0	0	$\{(1, 2)\}$
(0, 3, 6)	0	0	1	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

Table 6. Register Image Right After Coordinator Execution At Time 2

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	1	0	0	
(0, 3, 6)	0	0	0	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

1. Budget is consumed, and therefore future replenishment is added to *replenish*. Then server $(1, 1, 1)$ is selected as supplier, and its *carry* is decremented by 1. Table 8 shows the register image after the coordinator execution.

Table 7. Register Image Right After Component Loading At Time 3

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	1	1	0	
(0, 3, 6)	0	0	0	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

Table 8. Register Image Right After Coordinator Execution At Time 3

	budget	load	carry	replenish
(0, 0, 6)	1	0	0	
(1, 1, 1)	0	0	0	$\{(1, 5)\}$
(0, 3, 6)	0	0	0	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

At time 4, a job of task T_0 arrives. Therefore server $(0, 0, 6)$ is loaded by 1, as shown in Table 9. During the coordinator execution, budget is available for $(0, 0, 6)$ and consumed, future replenishment is stored, and the value of *carry* is incremented by 1. Then server $(0, 0, 6)$ is selected to supply, and its *carry* is decremented back to 0. Table 10 shows the register image after these executions.

Table 9. Register Image Right After Component Loading At Time 4

	budget	load	carry	replenish
(0, 0, 6)	1	1	0	
(1, 1, 1)	0	0	0	$\{(1, 5)\}$
(0, 3, 6)	0	0	0	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

Table 10. Register Image Right After Coordinator Execution At Time 4

	budget	load	carry	replenish
(0, 0, 6)	0	0	0	$\{(1, 68)\}$
(1, 1, 1)	0	0	0	$\{(1, 5)\}$
(0, 3, 6)	0	0	0	$\{(2, 64)\}$
(1, 3, 3)	0	0	1	$\{(1, 8)\}$
(2, 6, 6)	0	0	16	$\{(16, 64)\}$

It is noteworthy that a simple fixed-priority composition scheme cannot even compose C_0 and C_1 together for the following reason. Because of the short deadline of task T_0, C_0 must have the highest priority. Then there is a possibility that 3 continuous time units may be supplied to C_0, in which case task T_3 in C_1 may miss its deadline. The low composability is a result of not distinguishing the different types of workloads in C_0. In contrast, by Harmonic scheduler composition, C_0, C_1 and C_2 can be admitted one by one and served in the same time.

5.4 Analysis

If a component C_h is admitted by the coordinator, then the coordinator will supply resources to C_h according to the supply contract V_h. Assuming that there is a workload (k, l, w) in V_h, then a server (h, k, l) is established. Within any time interval of length m^l, up to w time units of supply may be loaded to the server, and every demand will obtain supply within m^k units of time since the demand is loaded. We call this the *responsiveness guarantee*. However, if

the accumulated load exceeds w time units within a time interval of length m^l, the server is *overloaded* and the responsiveness guarantee will not be provided anymore. The rationale here is that if the component breaks the supply contract by overloading, the coordinator cannot guarantee prompt supply. On the other hand, A non-overloaded server always provides the responsiveness guarantee, even when other servers (including other servers of the same component) are overloaded. We shall prove the responsiveness guarantee.

First, we prove that in a non-overloaded server, load never waits for budget.

Lemma 1 *For a non-overloaded server (h, k, l), load $\leq$ budget at any non-negative integer time t after budget replenishment.*

Proof: Base Case: At time 0, Register *budget* is initialized to w, and a non-overloading component loads less than or equal to w at time 0. The lemma is true.

Induction case: Assume that for any non-negative integer $t \leq n$, the lemma is true. We now prove that the lemma is still true at time $n+1$ by contradiction.

Assume the contrary: The value of *load* and the value of *budget* at time $n+1$ after replenishment is x and y, and $x > y$.

Let $n' = max(0, (n+1-m^l))$. Assume that the budget consumed after time n' but before or at time n is z, then $y + z = w$;

Because the lemma is true at time n', all loads arrived before or equal to time n' are carried before or at time n', so budget consumed between (n', n) is for load arrived after n' and before or at time n. Because the lemma is true for time n, *load* is decreased to 0 after the execution of the coordinator at time n. Therefore, the aggregate load after time n' and before or at time n is equal to the budget consumption during the the same interval of time, which is z.

Also, the aggregate arrival of load after time n but before or at time $n+1$ is x. The aggregate arrival of load after time n' and before or at time $n+1$ is $x + z$. Thus $x + z > y + z = w$, and the server is overloaded, a contradiction. ■

A non-negative integer time t is *class k un-carried* if all servers of class k or higher have zero value for *carry* before the coordinator execution at time t. At a class k un-carried time t, all previously loaded in-budget work for servers of class k or higher is completely supplied.

Lemma 2 *If t is a class k un-carried time, then there exists another class k un-carried time t' such that $t' \leq t + m^k$.*

Proof: According to the admission control algorithm, the aggregation of existing budget from all servers of class k or higher at time t before the coordinator execution and replenishment at or after time t and before $t + m^k$ will not exceed $P_k = m^k$. Therefore, the maximal aggregate value that can be added to *carry* of all servers of class k or higher will not exceed m^k. At any integer time t, if there exists a server of class k or higher with $carry > 0$, a supply is drawn from a server with class k or higher made and a *carry* is decreasing. If t' does not exist after time t and before time $t + m^k$, then *carry* is decreased by m^k at or after time t and before $t + m^k$, and time $t + m^k$ must be an class k un-carried time. Therefore the lemma holds. ■

Theorem 2 *If server (h, k, l) is not overloaded at any time, it provides the responsiveness guarantee.*

Proof: Time 0 is a class k un-carried time. According to Lemma 2, at any time t, there exists another un-carried time t' for class k before or at time $t+m^k$. According to Lemma 1, if component C_h adds *load* at time t, the complete load is moved to $carry$ at time t. Because $carry = 0$ at time t', the supply corresponding to the demand loaded at time t is made before time t'. Therefore responsiveness guarantee is maintained. ■.

The computational complexity of admission for a component C_h is bounded by $O(K \cdot |V_h|)$, where K is the maximal number of classes, and $|V_h|$ is the number of workloads in the contract which is bounded by K^2. The online coordinator overhead for each time unit is bounded by $O(n \cdot s)$, where n is the number of components and s is the maximal number of servers for a component which is bounded by K^2. Because the period of classes increases exponentially, K should be a small number.

6 Comparison with Related Work

There has been a significant amount of work on compositions in the last few years as has been pointed out in Section 1 of this paper. Instead of using EDF online for scheduling resource supply among components such as is in [2] and [4], our HCC approach distinguishes itself from these previous works by using a rate monotonic classification of workloads; the coordinator applies a fixed priority policy among workload classes. The urgency of workloads from components is expressed by their classes instead of explicit deadlines. The rate monotonic design of HCC makes admission control and budget management simple, yet maintains good composability.

Many hard and/or soft real-time scheduling approaches depend on a server budget to control the resource supply to a component to maintain a fair share. Total Bandwidth Server [6] is one example of this approach. Like servers, HCC also makes use of the budget idea. Because HCC is not deadline-based and temporal workload control depends totally on budget control, HCC does not require as much communication (e.g., deadlines of newly arrived jobs) between the system-level scheduler and the component schedulers and is hence a less costly and easier to implement budget-enforcement strategy.

Cayssials et al. proposed an approach to minimize the number of priorities in a rate-monotonic fixed priority scheme [1]. In their approach, multiple tasks are grouped into a class and scheduled on the same priority level. Although HCC solves a different problem and its design is significantly different from Cayssials et al, both algorithms exploit the idea of classification of workloads.

7 Future Work

Whereas the Harmonic Component Composition is a dynamic approach in which the coordinator does not depend on internal knowledge of components, we are

also investigating another approach to composition that improves composability and online resource supply efficiency by exploiting *a priori* knowledge of the components. Unlike the approach described in this paper, this alternative approach requires extensive offline computation. We believe that these two composition approaches span the two far ends of a wide spectrum of practical solutions for composing real-time schedulers. There is still much to be explored in the spectrum of solutions by a combination of the approaches. This is a subject for further investigation.

References

1. R.Cayssials, J. Orozco, J.Santos and R.Santos. Rate Monotonic Schedule of Real-Time Control Systems with the Minimum Number of Priority Levels, Euromicro Conference on Real Time Systems, pp. 54-59, 1999.
2. Z. Deng and J. Liu. Scheduling Real-Time Applications in an Open Environment. Real-Time Systems Symposium, pp. 308-319, December 1997.
3. G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems, Real-Time Systems Symposium, pp. 152-161, December 1995.
4. G. Lipari, J. Carpenter, S. Baruah. A Framework for Archieving Inter-Application Isolation in Multiprogrammed, Hard Real-Time Environment, Real-Time Systems Symposium, pp. 217-226, 2000.
5. A. K. Mok, X. Feng. Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds. Real-Time Systems Symposium, pp. 129-138, 2001.
6. M. Spuri, G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems, Real-Time Systems Journal, Vol,10, pp.179-210, 1996.
7. J. Regehr, J. A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. Real-Time Systems Symposium, pp. 3-14, 2001.
8. Duu-Chung Tsou. Execution Environment for Real-Time Rule-Based Decision Systems. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997.
9. W. Wang, A. K. Mok, Pre-Scheduling: Balancing Between Static and Dynamic Schedulers, UTCS Technical Report RTS-TR-02-01, 2002, http://www.cs.utexas.edu/users/mok/RTS/pubs.html.

Lazy Theorem Proving for Bounded Model Checking over Infinite Domains*

Leonardo de Moura, Harald Rueß, and Maria Sorea**

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{demoura, ruess, sorea}@csl.sri.com
http://www.csl.sri.com/

Abstract. We investigate the combination of propositional SAT checkers with domain-specific theorem provers as a foundation for bounded model checking over infinite domains. Given a program M over an infinite state type, a linear temporal logic formula φ with domain-specific constraints over program states, and an upper bound k, our procedure determines if there is a falsifying path of length k to the hypothesis that M satisfies the specification φ. This problem can be reduced to the satisfiability of Boolean constraint formulas. Our verification engine for these kinds of formulas is *lazy* in that propositional abstractions of Boolean constraint formulas are incrementally refined by generating lemmas on demand from an automated analysis of spurious counterexamples using theorem proving. We exemplify bounded model checking for timed automata and for RTL level descriptions, and investigate the lazy integration of SAT solving and theorem proving.

1 Introduction

Model checking decides the problem of whether a system satisfies a temporal logic property by exploring the underlying state space. It applies primarily to finite-state systems but also to certain infinite-state systems, and the state space can be represented in symbolic or explicit form. Symbolic model checking has traditionally employed a boolean representation of state sets using binary decision diagrams (BDD) [4] as a way of checking temporal properties, whereas explicit-state model checkers enumerate the set of reachable states of the system.

Recently, the use of Boolean satisfiability (SAT) solvers for linear-time temporal logic (LTL) properties has been explored through a technique known as *bounded model checking* (BMC) [7]. As with symbolic model checking, the state is encoded in terms

* This research was supported by SRI International internal research and development, the DARPA NEST program through Contract F33615-01-C-1908 with AFRL, and the National Science Foundation under grants CCR-00-86096 and CCR-0082560.

** Also affiliated with University of Ulm, Germany.

of booleans. The program is unrolled a bounded number of steps for some bound k, and an LTL property is checked for counterexamples over computations of length k. For example, to check whether a program with initial state I and next-state relation T violates the invariant Inv in the first k steps, one checks, using a SAT solver:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg Inv(s_0) \vee \ldots \vee \neg Inv(s_k)).$$

This formula is satisfiable if and only if there exists a path of length at most k from the initial state s_0 which violates the invariant Inv. For finite state systems, BMC can be seen as a complete procedure since the size of counterexamples is essentially bounded by the diameter of the system [3]. It has been demonstrated that BMC can be more effective in falsifying hypotheses than traditional model checking [7, 8].

It is possible to extend the range of BMC to infinite-state systems by encoding the search for a counterexample as a satisfiability problem for the logic of Boolean constraint formulas. For example, the BMC problem for timed automata can be captured in terms of a Boolean formula with linear arithmetic constraints. But the method presented here scales well beyond such simple arithmetic clauses, since the main requirement on any given constraint theory is the decidability of the satisfiability problem on conjunctions of atomic constraints. Possible constraint theories include, for example, linear arithmetic, bitvectors, arrays, regular expressions, equalities over terms with uninterpreted function symbols, and combinations thereof [20, 24].

Whereas BMC over finite-state systems deals with finding satisfying Boolean assignments, its generalization to infinite-state systems is concerned with satisfiability of Boolean constraint formulas. In initial experiments with PVS [21] strategies, based on a combination of BDDs for propositional reasoning and a variant of loop residue [27] for arithmetic, we were usually only able to construct counterexamples of small depths (≤ 5). Clearly, more specialized verification techniques are needed. Since BMC problems are often propositionally intensive, it seems to be more effective to augment SAT solvers with theorem proving capabilities, such as ICS [10], than add propositional search capabilities to theorem provers.

Here, we look at the specific combination of SAT solvers with decision procedures, and we propose a method that we call *lemmas on demand*, which invokes the theorem prover *lazily* in order to efficiently prune out spurious counterexamples, namely, counterexamples that are generated by the SAT solver but discarded by the theorem prover by interpreting the propositional atoms. For example, the SAT solver might yield the satisfying assignment $p, \neg q$, where the propositional variable p represents the atom $x = y$, and q represents $f(x) = f(y)$. A decision procedure can easily detect the inconsistency in this assignment. More importantly, it can be used to generate a set of conflicting assignments that can be used to construct a lemma that further constrains the search. In the above example, the lemma $\neg p \vee q$ can be added as a new clause in the input to the SAT solver. This process of refining Boolean formulas is similar in spirit to the refinement of abstractions based on the analysis of spurious counterexamples or failed proof attempts [26, 25, 6, 16, 9, 14, 17].

From a set of inconsistent constraints in a spurious counterexample we obtain an *explanation* as an overapproximation of the minimal, inconsistent subset of these constraints. The smaller the explanation that is generated from a spurious counterexample,

the greater the pruning in the subsequent search. In this way, the computation of explanations accelerates the convergence of our procedure.

Altogether, we present a method for bounded model checking over infinite-state systems that consists of:

- A reduction to the satisfiability problem for Boolean constraint formulas.
- A lazy combination of SAT solving and theorem proving.
- An efficient method for constructing small explanations.

In general, BMC over infinite-state systems is not complete, but we obtain a completeness result for BMC problems with invariant properties. The main condition on constraints is that the satisfiability of the conjunction of constraints is decidable. Thus, our BMC procedure can be applied to infinite-state systems even when the (more) general model-checking problem is undecidable.

The paper is structured as follows. In Section 2 we provide some background material on Boolean constraints. Section 3 lays the foundation of a refinement-based satisfiability procedure for Boolean constraint logic. Next, Section 4 presents the details of BMC over domain-specific constraints, and Section 5 discusses some simple examples for BMC over clock constraints and the theory of bitvectors. In Section 6 we experimentally investigate various design choices in lazy integrations of SAT solvers with theorem proving. Finally, in Sections 7 and 8 we compare with related work and we draw conclusions.

2 Background

A set of variables $V := \{x_1, \ldots, x_n\}$ is said to be typed if there are nonempty sets D_1 through D_n and a *type assignment* τ such that $\tau(x_i) = D_i$. For a set of typed variables V, a *variable assignment* is a function ν from variables $x \in V$ to an element of $\tau(x)$.

Let V be a set of typed variables and L be an associated logical language. A set of constraints in L is called a *constraint theory* $\mathcal{C}$ if it includes constants $true$, $false$ and if it is closed under negation; a subset of $\mathcal{C}$ of constraints with free variables in $V' \subseteq V$ is denoted by $\mathcal{C}(V')$. For $c \in \mathcal{C}$ and ν an assignment for the free variables in c, the value of the predicate $[\![c]\!]_\nu$ is called the *interpretation* of c w.r.t. ν. Hereby, $[\![true]\!]_\nu$ ($[\![false]\!]_\nu$) is assumed to hold for all (for no) ν, and $[\![\neg c]\!]_\nu$ holds iff $[\![c]\!]_\nu$ does not hold. A set of constraints $C \subseteq \mathcal{C}$ is said to be *satisfiable* if there exists a variable assignment ν such that $[\![c]\!]_\nu$ holds for every c in C; otherwise, C is said to be *unsatisfiable*. Furthermore, a function $\mathcal{C}$-*sat*(C) is called a $\mathcal{C}$-satisfiability solver if it returns $\perp$ if the set of constraints C is unsatisfiable and a satisfying assignment for C otherwise.

For a given theory $\mathcal{C}$, the set of *boolean constraints* $\mathsf{Bool}(\mathcal{C})$ includes all constraints in $\mathcal{C}$ and it is closed under conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. The notions of satisfiability, inconsistency, satisfying assignment, and satisfiability solver are homomorphically lifted to the set of boolean constraints in the usual way. If $V = \{p_1, \ldots, p_n\}$ and the corresponding type assignment $\tau(p_i)$ is either true or false, then $\mathsf{Bool}(\{true, false\} \cup V)$ reduces to the usual notion of Boolean logic with propositional variables $\{p_1, \ldots, p_n\}$. We call a Boolean satisfiability solver also a SAT solver. N-ary disjunctions of constraints are also referred to as *clauses*, and a formula $\varphi \in$

$\mathsf{Bool}(\mathcal{C}(V))$ is in *conjunctive normal form* (CNF) if it is an n-ary conjunction of clauses. There is a linear-time satisfiability-preserving transformation into CNF [22].

3 Lazy Theorem Proving

Satisfiability solvers for propositional constraint formulas can be obtained from the combination of a propositional SAT solver with decision procedures simply by converting the problem into disjunctive normal form, but the result is prohibitively expensive. Here, we lay out the foundation of a lazy combination of SAT solvers with constraint solvers based on an incremental refinement of Boolean formulas. We restrict our analysis to formulas in CNF, since most modern SAT solvers expect their input to be in this format.

Translation schemes between propositional formulas and Boolean constraint formulas are needed. Given a formula φ such a correspondence is easily obtained by abstracting constraints in φ with (fresh) propositional variables. More formally, for a formula $\varphi \in \mathsf{Bool}(\mathcal{C})$ with atoms $C = \{c_1, \ldots, c_n\} \in \mathcal{C}$ and a set of propositional variables $P = \{p_1, \ldots, p_n\}$ not occurring in φ, the mapping α from Boolean formulas over $\{c_1, \ldots, c_n\}$ to Boolean formulas over P is defined as the homomorphism induced by $\alpha(c_i) = p_i$. The inverse γ of such an abstraction mapping α simply replaces propositional variables p_i with their associated constraints c_i. For example, the formula $\varphi \equiv f(x) \neq x \wedge f(f(x)) = x$ over equalities of terms with uninterpreted function symbols determines the function α with, say, $\alpha(f(x) \neq x) = p_1$ and $\alpha(f(f(x)) = x) = p_2$; thus $\alpha(\varphi) = p_1 \wedge p_2$. Moreover, a Boolean assignment $\nu : P \to \{\mathit{true}, \mathit{false}\}$ induces a set of constraints

$$\gamma(\nu) \equiv \{c \in \mathcal{C} \mid \exists i.\ \text{if } \nu(p_i) = \mathit{true} \text{ then } c = \gamma(p_i) \text{ else } c = \neg\gamma(p_i)\}\ .$$

Now, given a Boolean variable assignment ν such that $\nu(p_1) = \mathit{false}$ and $\nu(p_2) = \mathit{true}$, $\gamma(\nu)$ is the set of constraints $\{f(x) = x, f(f(x)) = x\}$. A consistent set of constraints C determines a set of assignments. For choosing an arbitrary, but fixed assignment from this set, we assume as given a function $\mathit{choose}(C)$.

Theorem 1. Let $\varphi \in \mathsf{Bool}(\mathcal{C})$ be a formula in CNF, $\mathcal{L}$ be the literals in $\alpha(\varphi)$, and $I(\varphi) := \{L \subseteq \mathcal{L} \mid \gamma(L) \text{ is } \mathcal{C}\textit{-inconsistent}\}$ be the set of $\mathcal{C}$-inconsistencies for φ; then: φ is *$\mathcal{C}$-satisfiable* iff the following Boolean formula is satisfiable:

$$\alpha(\varphi) \wedge \Big(\bigwedge_{\{l_1,\ldots,l_n\} \in I(\varphi)} (\neg l_1 \vee \ldots \vee \neg l_n)\Big).$$

Thus, every $\mathsf{Bool}(\mathcal{C})$ formula can be transformed into an equisatisfiable Boolean formula as long as the consistency problem for sets of constraints in $\mathcal{C}$ is decidable. This transformation enables one to use off-the-shelf satisfiability checkers to determine the satisfiability of Boolean constraint formulas. On the other hand, the set of literals is exponential in the number of variables and, therefore, an exponential number of *$\mathcal{C}$-inconsistency* checks is required in the worst case. It has been observed, however, that in many cases only small fragments of the set of *$\mathcal{C}$-inconsistencies* are needed.

$\mathbf{sat}(\varphi)$
  $p := \alpha(\varphi)$;
  loop
    $\nu := \mathcal{B}\text{-}sat(p)$;
    if $\nu = \bot$ **then return** $\bot$;
    if $\mathcal{C}\text{-}sat(\gamma(\nu)) \neq \bot$ **then return** $choose(\gamma(\nu))$;
    $I := \bigvee_{c \in \gamma(\nu)} \neg\alpha(c)$; $p := p \wedge I$
  endloop

Fig. 1. Lazy theorem proving for $\mathsf{Bool}(\mathcal{C})$.

Starting with $p = \alpha(\varphi)$, the procedure $\mathbf{sat}(\varphi)$ in Figure 1 realizes a guided enumeration of the set of *$\mathcal{C}$-inconsistencies*. In each loop, the SAT solver $\mathcal{B}\text{-}sat$ suggests a candidate assignment ν for the Boolean formula p, and the satisfiability solver $\mathcal{C}\text{-}sat$ for $\mathcal{C}$ checks whether the corresponding set of constraints $\gamma(\nu)$ is consistent. Whenever this consistency check fails, p is refined by adding a Boolean analogue I of this inconsistency, and $\mathcal{B}\text{-}sat$ is applied to suggest a new candidate assignment for the refined formula $p \wedge I$. This procedure terminates, since, in every loop, I is not subsumed by p, and there are only a finite number of such strengthenings.

Corollary 1. $\mathbf{sat}(\varphi)$ in Figure 1 is a satisfiability solver for $\mathsf{Bool}(\mathcal{C})$ formulas in CNF.

We list some essential optimizations. If the variable assignments returned by the SAT solver are partial in that they include *don't care* values, then the number of argument constraints to $\mathcal{C}\text{-}sat$ can usually be reduced considerably. The use of don't care values also speeds up convergence, since more general lemmas are generated. Now, assume a function $explain(C)$, which, for an inconsistent set of constraints C, returns a minimal number of inconsistent constraints in C or a "good" overapproximation thereof. The use of $explain(C)$ instead of the stronger C obviously accelerates the procedure. We experimentally analyze these efficiency issues in Section 6.

4 Infinite-State BMC

Given a BMC problem for an infinite-state program, an LTL formula with constraints, and a bound on the length of counterexamples to be searched for, we describe a sound reduction to the satisfiability problem of Boolean constraint formulas and we show completeness for invariant properties. The encoding of transition relations follows the now-standard approach already taken in [13]. Whereas in [7] LTL formulas are translated directly into propositional formulas, we use Büchi automata for this encoding. This simplifies substantially the notations and the proofs, but a direct translation can sometimes be more succinct in the number of variables needed. We use the common

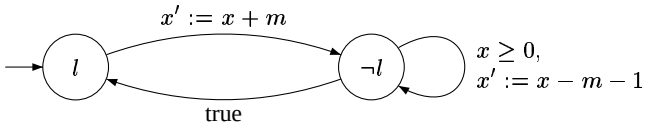

Fig. 2. The *simple* example.

notions for finite automata over finite and infinite words, and we assume as given a constraint theory $\mathcal{C}$ with a satisfiability solver.

Typed variables in $V := \{x_1, \ldots, x_n\}$ are also called *state variables*, and a *program state* is a variable assignment over V. A pair $\langle I, T \rangle$ is a *$\mathcal{C}$-program* over V if $I \in \mathsf{Bool}(\mathcal{C}(V))$ and $T \in \mathsf{Bool}(\mathcal{C}(V \cup V'))$, where V' is a primed, disjoint copy of V. I is used to restrict the set of initial program states, and T specifies the transition relation between states and their successor states. The set of $\mathcal{C}$-programs over V is denoted by $\mathsf{Prg}(\mathcal{C}(V))$. The semantics of a program P is given in terms of a *transition system* M in the usual way, and, by a slight abuse of notation, we sometimes write M for both the program and its associated transition system. The system depicted in Figure 2, for example, is expressed in terms of the program $\langle I, T \rangle$ over $\{x, l\}$, where the counter x is interpreted over the integers and the variable l for encoding locations is interpreted over the Booleans (the n-ary connective $\otimes$ holds iff exactly one of its arguments holds).

$$\begin{aligned} I(x,l) &:= x \geq 0 \wedge l \\ T(x,l,x',l') &:= (l \wedge x' = x + m \wedge \neg l') \otimes \\ &\quad (\neg l \wedge x \geq 0 \wedge x' = x - m - 1 \wedge \neg l') \otimes (\neg l \wedge x' = x \wedge l') \end{aligned}$$

Initially, the program is in location l and x is greater than or equal to 0, and the transitions in Figure 2 are encoded by a conjunction of constraints over the current state variables x, l and the next state variables x', l'.

The formulas of the *constraint linear temporal logic* $\mathsf{LTL}(\mathcal{C})$ (in negation normal form) are linear-time temporal logic formulas with the usual "next", "until", and "release" operators, and constraints $c \in \mathcal{C}$ as atoms.

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid c \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \mid \varphi_1\,\mathbf{R}\,\varphi_2$$

The formula $\mathbf{X}\,\varphi$ holds on some path π iff φ holds in the second state of π. $\varphi_1\,\mathbf{U}\,\varphi_2$ holds on π if there is a state on the path where φ_2 holds, and at every preceding state on the path φ_1 holds. The release operator $\mathbf{R}$ is the logical dual of $\mathbf{U}$. It requires that φ_2 holds along the path up to and including the first state, where φ_1 holds. However, φ_1 is not required to hold eventually. The derived operators $\mathbf{F}\,\varphi = \mathit{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi = \mathit{false}\,\mathbf{R}\,\varphi$ denote "eventually φ" and "globally φ". Given a program $M \in \mathsf{Prg}(\mathcal{C})$ and a path π in M, the satisfiability relation $M, \pi \models \varphi$ for an $\mathsf{LTL}(\mathcal{C})$ formula φ is given in the usual way with the notable exception of the case of constraint formulas c. In this case, $M, \pi \models c$ if and only if c holds in the start state of π. Assuming the notation above, the *$\mathcal{C}$-model checking problem* $M \models \varphi$ holds iff for all paths $\pi = s_0, s_1, \ldots$ in M with $s_0 \in I$ it is the case that $M, \pi \models \varphi$. Given a bound k, a program $M \in \mathsf{Prg}(\mathcal{C})$ and a formula $\varphi \in \mathsf{LTL}(\mathcal{C})$ we now consider the problem of constructing a formula $[\![M, \varphi]\!]_k \in \mathsf{Bool}(\mathcal{C})$, which is satisfiable if and only if there is a counterexample of

length k for the $\mathcal{C}$-model checking problem $M \models \varphi$. This construction proceeds as follows.

1. Definition of $[\![M]\!]_k$ as the unfolding of the program M up to step k from initial states (this requires k disjoint copies of V).
2. Translation of $\neg\varphi$ into a corresponding Büchi automaton $\mathcal{B}_{\neg\varphi}$ whose language of accepting words consists of the satisfying paths of $\neg\varphi$.
3. Encoding of the transition system for $\mathcal{B}_{\neg\varphi}$ and the Büchi acceptance condition as a Boolean formula, say $[\![\mathcal{B}]\!]_k$.
4. Forming the conjunction $[\![M, \varphi]\!]_k := [\![\mathcal{B}]\!]_k \wedge [\![M]\!]_k$.
5. A satisfying assignment for the formula $[\![M, \varphi]\!]_k$ induces a counterexample of length k for the model checking problem $M \models \varphi$.

Definition 1 (Encoding of $\mathcal{C}$-*Programs*). The encoding $[\![M]\!]_k$ of the kth unfolding of a $\mathcal{C}$-*program* $M = \langle I, T\rangle$ in $\mathsf{Prg}(\mathcal{C}(\{x_1, \ldots, x_n\}))$ is given by the $\mathsf{Bool}(\mathcal{C})$ formula $[\![M]\!]_k$.

$$
\begin{aligned}
I_0(x[0]) &:= I\langle\{x_i \mapsto x_i[0] \mid x_i \in V\}\rangle \\
T_j(x[j], x[j+1]) &:= T\langle\{x_i \mapsto x_i[j] \mid x_i \in V\} \cup \{x_i' \mapsto x_i[j+1] \mid x_i \in V\}\rangle \\
[\![M]\!]_k &:= I_0(x[0]) \wedge \bigwedge_{j=0}^{k-1} T_j(x[j], x[j+1])
\end{aligned}
$$

where $\{x_i[j] \mid 0 \leq j \leq k\}$ is a family of typed variables for encoding the state of variable x_i in the jth step, $x[j]$ is used as an abbreviation for $x_1[j], \ldots, x_n[j]$, and $T\langle x_i \mapsto x_i[j]\rangle$ denotes simultaneous substitution of x_i by $x_i[j]$ in formula T.

A two-step unfolding of the *simple* program in Figure 2 is encoded by $[\![simple]\!]_2 := I_0 \wedge T_0 \wedge T_1$ (*).

$$
\begin{aligned}
I_0 &:= x[0] \geq 0 \wedge l[0] \\
T_0 &:= (\, l[0] \wedge (x[1] = x[0] + m) \wedge \neg l[1] \,) \otimes \\
&\quad (\, \neg l[0] \wedge (x[0] \geq 0) \wedge (x[1] = x[0] - m - 1) \wedge \neg l[1] \,) \otimes \\
&\quad (\, \neg l[0] \wedge (x[1] = x[0]) \wedge l[1] \,) \\
T_1 &:= (\, l[1] \wedge (x[2] = x[1] + m) \wedge \neg l[2] \,) \otimes \\
&\quad (\, \neg l[1] \wedge (x[1] \geq 0) \wedge (x[2] = x[1] - m - 1) \wedge \neg l[2] \,) \otimes \\
&\quad (\, \neg l[1] \wedge (x[2] = x[1]) \wedge l[2] \,)
\end{aligned}
$$

The translation of linear temporal logic formulas into a corresponding Büchi automaton is well-studied in the literature [11] and does not require additional explanation. Notice, however, that the translation of $\mathsf{LTL}(\mathcal{C})$ formulas yields Büchi automata with $\mathcal{C}$-*constraints* as labels. Both the resulting transition system and the bounded acceptance test based on the detection of reachable cycles with at least one final state can easily be encoded as $\mathsf{Bool}(\mathcal{C})$ formulas.

8

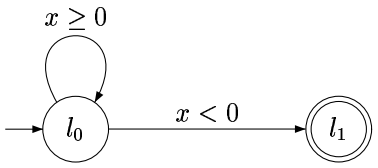

Fig. 3. Automaton for $\mathbf{F}\,(x < 0)$.

Definition 2 (Encoding of Büchi Automata). Let $V = \{x_1, \ldots, x_n\}$ be a set of typed variables, $\mathcal{B} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ be a Büchi automaton with labels Σ in $\mathsf{Bool}(\mathcal{C})$, and pc be a variable (not in V), which is interpreted over the finite set of locations Q of the Büchi automaton. For a given integer k, we obtain, as in Definition 1, families of variables $x_i[j]$, $pc[j]$ $(1 \leq i \leq n,\ 0 \leq j \leq k)$ for representing the jth state of $\mathcal{B}$ in a run of length k. Furthermore, the transition relation of $\mathcal{B}$ is encoded in terms of the *$\mathcal{C}$-program* $\mathcal{B}_M$ over the set of variables $\{pc\} \cup V$, and $[\![\mathcal{B}_M]\!]_k$ denotes the encoding of this program as in Definition 1. Now, given an encoding of the acceptance condition

$$acc(\mathcal{B})_k \;:=\; \bigvee_{j=0}^{k-1} \Big(pc[k] = pc[j] \wedge \bigwedge_{v=1}^{n} x_v[k] = x_v[j] \wedge \Big(\bigvee_{l=j+1}^{k} \bigvee_{f \in F} pc[l] = f \Big) \Big)$$

the k-th unfolding of $\mathcal{B}$ is defined by $[\![\mathcal{B}]\!]_k := [\![\mathcal{B}_M]\!]_k \wedge acc(\mathcal{B})_k$.

An $\mathsf{LTL}(\mathcal{C})$ formula is said to be **R**-free (**U**-free) iff there is an equivalent formula (in negation normal form) not containing the operator **R** (**U**). Note that **U**-free formulas correspond to the notion of *syntactic safety formulas* [28, 15]. Now, it can be directly observed from the semantics of $\mathsf{LTL}(\mathcal{C})$ formulas that every **R**-free formula can be translated into an automaton over finite words that accepts a prefix of all infinite paths satisfying the given formula.

Definition 3. Given an automaton $\mathcal{B}$ over finite words and the notation as in Definition 2, the encoding of the k-ary unfolding of $\mathcal{B}$ is given by $[\![\mathcal{B}_M]\!]_k \wedge acc(\mathcal{B})_k$ with the acceptance condition

$$acc(\mathcal{B})_k \;:=\; \bigvee_{j=0}^{k} \bigvee_{f \in F} pc[j] = f\ .$$

Consider the problem of finding a counterexample of length $k = 2$ to the hypothesis that our running example in Figure 2 satisfies $\mathbf{G}\,(x \geq 0)$. The negated property $\mathbf{F}\,(x < 0)$ is an **R**-free formula, and the corresponding automaton $\mathcal{B}$ over finite words is displayed in Figure 3 (l_1 is an accepting state.). This automaton is translated, according to Definition 3, into the formula

$$[\![\mathcal{B}]\!]_2 := I(\mathcal{B}) \wedge T_0(\mathcal{B}) \wedge T_1(\mathcal{B}) \wedge acc(\mathcal{B})_2\ . \qquad (**)$$

The variables $pc[j]$ and $x[j]$ $(j = 0, 1, 2)$ are used to represent the first three states in a run.

$$\begin{aligned}
I(\mathcal{B}) &:= pc[0] = l_0 \\
T_0(\mathcal{B}) &:= (pc[0] = l_0 \wedge x[0] \geq 0 \wedge pc[1] = l_0) \otimes (pc[0] = l_0 \wedge x[0] < 0 \wedge pc[1] = l_1) \\
T_1(\mathcal{B}) &:= (pc[1] = l_0 \wedge x[1] \geq 0 \wedge pc[2] = l_0) \otimes (pc[1] = l_0 \wedge x[1] < 0 \wedge pc[2] = l_1) \\
acc(\mathcal{B})_2 &:= pc[0] = l_1 \vee pc[1] = l_1 \vee pc[2] = l_1
\end{aligned}$$

The bounded model checking problem $[\![simple]\!]_2 \wedge [\![\mathcal{B}]\!]_2$ for the *simple* program is obtained by conjoining the formulas $(*)$ and $(**)$. Altogether, we obtain the counterexample $(0, l) \rightarrow (m, \neg l) \rightarrow (-1, l)$ of length 2 for the property $\mathbf{G}\,(x \geq 0)$.

Theorem 2 (Soundness). Let $M \in \mathsf{Prg}(\mathcal{C})$ and $\varphi \in \mathsf{LTL}(\mathcal{C})$. If there exists a natural number k such that $[\![M, \varphi]\!]_k$ is satisfiable, then $M \not\models \varphi$.
Proof sketch. If $[\![M, \varphi]\!]_k$ is satisfiable, then so are $[\![\mathcal{B}]\!]_k$ and $[\![M]\!]_k$. From the satisfiability of $[\![\mathcal{B}]\!]_k$ it follows that there exists a path in the Büchi automaton $\mathcal{B}$ that accepts the negation of the formula φ.

In general, BMC over infinite-state systems is not complete. Consider, for example, the model checking problem $M \models \varphi$ for the program $M = \langle I, T \rangle$ over the variable $V = \{x\}$ with $I = (x = 0)$ and $T = (x' = x + 1)$ and the formula $\varphi = \mathbf{F}\,(x < 0)$. M can be seen as a one-counter automaton, where initially the value of the counter x is 0, and in every transition the value of x is incremented by 1. Obviously, it is the case that $M \not\models \varphi$, but there exists no $k \in I\!N$ such that the formula $[\![M, \varphi]\!]_k$ is satisfiable. Since $\neg\varphi$ is not an **R**-free formula, the encoding of the Büchi automaton $\mathcal{B}_k$ must contain, by Definition 2, a finite accepting cycle, described by $pc[k] = pc[0] \wedge x[k] = x[0]$ or $pc[k] = pc[1] \wedge x[k] = x[1]$ etc. Such a cycle, however, does not exist, since the program M contains only one noncycling, infinite path, where the value of x increases in every step, that is $x[i + 1] = x[i] + 1$, forall $i \geq 0$.

Theorem 3 (Completeness for Finite States). Let M be a *$\mathcal{C}$-program* with a finite set of reachable states, φ be an $\mathsf{LTL}(\mathcal{C})$ formula φ, and k be a given bound; then: $M \models\!\!\!/\ \varphi$ implies $\exists k \in I\!N.\, [\![M, \varphi]\!]_k$ is satisfiable.
Proof sketch. If $M \not\models \varphi$, then there is a path in M that falsifies the formula. Since the set of reachable states is finite, there is a finite k such that $[\![M, \varphi]\!]_k$ is satisfiable by construction.

For a **U**-free formula φ, the negation $\neg\varphi$ is **R**-free and can be encoded in terms of an automaton over finite words. Therefore, by considering only **U**-free properties one gets completeness also for programs with an infinite set of reachable states. A particularly interesting class of **U**-free formulas are invariant properties.

Theorem 4 (Completeness for Syntactic Safety Formulas). Let M be a *$\mathcal{C}$-program*, $\varphi \in \mathsf{LTL}(\mathcal{C})$ be a **U**-free property, and k be some given integer bound. Then $M \models\!\!\!/\ \varphi$ implies $\exists k \in I\!N.\, [\![M, \varphi]\!]_k$ is satisfiable.
Proof sketch. If $M \not\models \varphi$ and φ is **U**-free then there is a finite prefix of a path of M that falsifies φ. Thus, by construction of $[\![M, \varphi]\!]_k$, there is a finite k such that $[\![M, \varphi]\!]_k$ is satisfiable.

This completeness result can easily be generalized to all safety properties [15] by observing that the prefixes violated by these properties can also be accepted by an automaton on finite words.

5 Examples

We demonstrate BMC over clock constraints and the theory of bitvectors by means of some simple but, we think, illustrative examples.

10

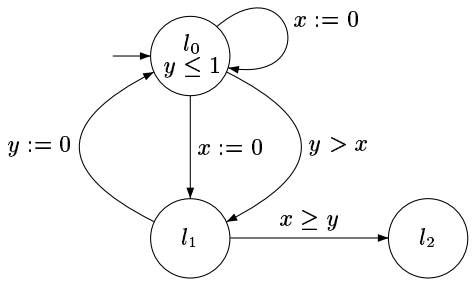

Fig. 4. Timed automata example.

The timed automaton [1] in Figure 4 has two real-valued clocks x, y, the transitions are decorated with clock constraints and clock resets, and the invariant $y \leq 1$ in location l_0 specifies that the system may stay in l_0 only as long as the value of y does not exceed 1. The transitions can easily be described in terms of a program with linear arithmetic constraints over states (pc, x, y), where pc is interpreted over the set of locations $\{l_0, l_1, l_2\}$ and the clock variables x, y are interpreted over $\mathbb{R}_0^+$. Here we show only the encoding of the time $delay$ steps.

$$delay(pc, x, y, pc', x', y') :=$$
$$\exists\, \delta \geq 0.\, ((pc = l_0 \Rightarrow y' \leq 1) \wedge (x' = x + \delta) \wedge (y' = y + \delta) \wedge (pc' = pc)).$$

This relation can easily be transformed into an equivalent quantifier-free formula. Now, assume the goal of falsifying the hypothesis that the timed automaton in Figure 4 satisfies the $\mathsf{LTL}(\mathcal{C})$ property $\varphi = (\mathbf{G}\, \neg l_2)$, that is, the automaton never reaches location l_2. Using the BMC procedure over linear arithmetic constraints one finds the counterexample

$$(l_0, x = 0, y = 0) \rightarrow (l_1, x = 0, y = 0) \rightarrow (l_2, x = 0, y = 0)$$

of length 2. By using Skolemization of the delay step δ instead of quantifier elimination, explicit constraints are synthesized for the corresponding delay steps in countertraces.

Now, we examine BMC over a theory $\mathcal{B}$ of bitvectors by encoding the shift register example in [3] as follows.

$$I_{BS}(x_n) := true \qquad T_{BS}(x_n, y_n) := (y_n = x_n[1 : n-1] \star 1_1)$$

The variables x_n and y_n are interpreted over bitvectors of length n, $x_n[1 : n-1]$ denotes extraction of bits 1 through $n-1$, $\star$ denotes concatenation, and 0_n (1_n) is the constant bitvector of length n with all bits set to zero (one). In the initial state the content of the register x_n is arbitrary. Given the $\mathsf{LTL}(\mathcal{B})$ property $\varphi = \mathbf{F}\, (x_n = 0_n)$ and $k = 2$ the corresponding BMC problem reduces to showing satisfiability of the $\mathsf{Bool}(\mathcal{B})$ formula

$$(x_1 = x_0[1 : n-1] \star 1_1) \wedge (x_2 = x_1[1 : n-1] \star 1_1) \wedge$$
$$(x_0 \neq 0_n \vee x_1 \neq 0_n \vee x_2 \neq 0_n) \wedge (x_0 = x_2 \vee x_1 = x_2).$$

The variables x_0, x_1, x_2 are interpreted over bitvectors of size n, since they are used to represent the first three states in a run of the shift register. The satisfiability of this

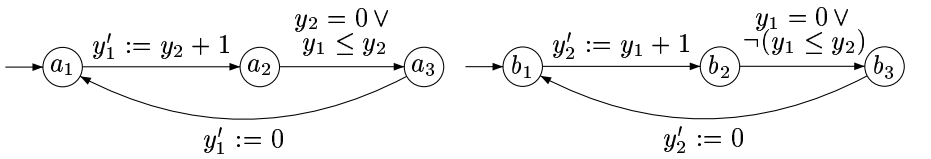

Fig. 5. Bakery Mutual Exclusion Protocol.

formula is established by choosing all unit literals to be true. Using theory-specific canonization (rewrite) steps for the bitvector theory $\mathcal{B}$ [18], we obtain an equation between variables x_2 and x_0.

$$x_2 \;=\; x_1[1:n-1] \star 1_1 \;=\; (x_0[1:n-1] \star 1_1)[1:n-1] \star 1_1 \;=\; x_0[2:n-1] \star 1_2$$

This canonization step corresponds to a symbolic simulation of depth 2 of the synchronous circuit. Now, in case the SAT solver decides the equation $x_0 = x_2$ to be true, the bitvector decision procedures are confronted with solving the equality $x_0 = x_0[2:n-1] \star 1_2$. The most general solution for x_0 is obtained using the solver in [18] and, by simple backsubstitution, one gets a satisfying assignment for x_0, x_1, x_2, which serves as a counterexample for the assertion that the shift register eventually is zero. The number of case splits is linear in the bound k, and, by leaving the word size uninterpreted, our procedure invalidates a family of shift registers without runtime penalties.

6 Efficiency Issues

The purpose of the experiments in this section is to identify useful concepts and techniques for obtaining efficient implementations of the lazy theorem proving approach. For these experiments we implemented several refinements of the basic lazy theorem proving algorithm from Section 3, using SAT solvers such as Chaff [19] and ICS [10] for deciding linear arithmetic constraints. These programs either returns $\perp$ in case the input Boolean constraint problem is unsatisfiable or an assignment for the variables. We describe some of our experiments using the Bakery mutual exclusion protocol (see Figure 5). Usually, the y_i counters are initialized with 0, but here we simultaneously consider a family of Bakery algorithms by relaxing the condition on initial values of the counters to $y_1 \geq 0 \wedge y_2 \geq 0$. Our experiments represent worst-case scenarios in that the corresponding BMC problems are all unsatisfiable. Thus, unsatisfiability of the BMC formula for a given k corresponds to a verification of the mutual exclusion property for paths of length $\leq k$.

Initial experiments with a direct implementation of the refinement algorithm in Figure 1 clearly show that this approach quickly becomes impractical. We identified two main reasons for this inefficiency.

First, for the interleaving semantics of the Bakery processes, usually only a small subset of assignments is needed for establishing satisfiability. This can already be demonstrated using the *simple* example in Figure 2. Suppose a satisfying assignment ν (counterexample) corresponding to executing the transition $l \longrightarrow \neg l$ with $x' = x + m$ in the

first step; that is, $[\![l[0]]\!]_\nu$, $[\![x[1] = x[0] + m]\!]_\nu$ and $[\![\neg l[1]]\!]_\nu$ hold. Clearly, the value of the literals $x[0] \geq 0$, $x[1] = x[0] - m - 1$, and $x[1] = x[0]$ are *don't cares*, since they are associated with some other transition. Overly eager assignment of truth values to these constraints results in useless search. For example, if $[\![x[1] = x[0]]\!]_\nu$ holds, then an inconsistency is detected, since $m > 0$, and $x[1] = x[0] + m = x[0]$. Consequently, the assignment ν is discarded and the search continues. To remedy the situation we analyze the structure of the formula before converting it to CNF, and use this information to assign *don't care* values to literals corresponding to unfired transitions in each step.

Second, the convergence of the refinement process must be accelerated by finding concise overapproximations *explain*(C) of the minimal set of inconsistent constraints C corresponding to a given Boolean assignment. There is an obvious trade-off between the conciseness of this approximation and the cost for computing it. We are proposing an algorithm for finding such an overapproximation based on rerunning the decision procedures $O(m \times n)$ times, where m is some given upper bound on the number of iterations (see below) and n is the number of given constraints.

The run in Figure 6 illustrates this procedure. The constraints in Figure 6.(a) are asserted to ICS from left-to-right. Since ICS detects a conflict when asserting $y_6 \leq 0$, this constraint is in the minimal inconsistent set. Now, an overapproximation of the minimal inconsistent sets is produced by connecting constraints with common variables (Figure 6.(a)). This overapproximation is iteratively refined by collecting the constraints in an array as illustrated in Figure 6.(b). Configurations consist of triples (C, l, h), where C is a set of constraints guaranteed to be in the minimal inconsistent set, and the integers l, h are the lower and upper bounds of constraint indices still under consideration. The initial configuration in our example is $(\{y_6 \leq 0\},\ 0,\ 3)$. In each refinement step, we maintain the invariant that $C \cup \{array[i] \mid l \leq i \leq h\}$ is inconsistent. Given a configuration (C, l, h), individual constraints of index between l and h are added to C until an inconsistency is detected. In the first iteration of our running example, we process constraints from right-to-left, and an inconsistency is only detected when processing $y_5 > 0$. The new configuration $(\{y_6 \leq 0, y_5 > 0\}, 1, 3)$ is obtained by adding this constraint to the set of constraints already known to be in a minimal inconsistent set, by leaving h unchanged, and by setting l to the increment of the index of the new constraint. The order in which constraints are asserted is inverted after each iteration. Thus, in the next step in our example, we successively add constraints between 1 and 3 from left-to-right to the set $\{y_6 \leq 0, y_5 > 0\}$. An inconsistency is first detected when asserting $y_6 = y_5$ to this set, and the new configuration is obtained as $(\{y_6 \leq 0, y_5 > 0, y_6 = y_5\}, 1, 1)$, since the lower bound l is now left unchanged and the upper bound is set to the decrement of the index of the constraint for which the inconsistency has been detected. The procedure terminates if C in the current configuration is inconsistent or after m refinements. In our example, two refinement steps yield the minimal inconsistent set $\{y_5 > 0, y_6 = y_5, y_6 \leq 0\}$. In general, the number of assertions is linear in the number of constraints, and the algorithm returns the exact minimal set if its cardinality is less than or equal to the upper bound m of iterations.

Given these refinements to the satisfiability algorithm in Figure 1, we implemented an *offline* integration of Chaff with ICS, in which the SAT solver and the decision procedures are treated as black boxes, and both procedures are restarted in each lazy refine-

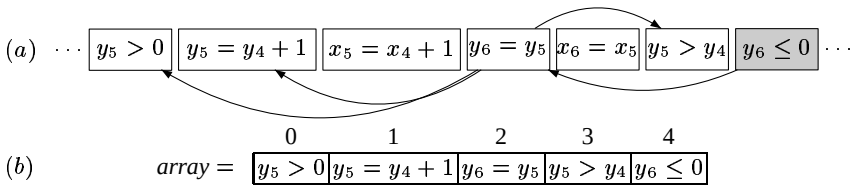

Fig. 6. Trace for linear time *explain* function.

ment step. Table 1 includes some statistics for three different configurations depending on whether *don't care* processing or the linear *explain* are enabled. For each configuration, we list the total time (in seconds) and the number of conflicts detected by the decision procedure. This table indicates that the effort of assigning don't care values

	don't cares, no explain		no don't cares, explain		don't cares, explain	
depth	time	conflicts	time	conflicts	time	conflicts
5	0.71	66	45.23	577	0.31	16
6	2.36	132	83.32	855	0.32	18
7	12.03	340	286.81	1405	1.75	58
8	56.65	710	627.90	1942	2.90	73
9	230.88	1297	1321.57	2566	8.00	105
10	985.12	2296	-	-	15.28	185
15	-	-	-	-	511.12	646

Table 1. Offline lazy theorem proving ('-' is time ≥ 1800 secs).

depending on the asynchronous nature of the program and the use of explain functions significantly improves performance.

Recall that the experiments so far represent worst-case scenarios in that the given formulas are unsatisfiable. For BMC problems with counterexamples, however, our procedure usually converges much faster. Consider, for example the mutual exclusion problem of the Bakery protocol with a guard $y_1 \geq y_2 - 1$ instead of $\neg(y_1 \leq y_2)$. The corresponding counterexample for $k = 5$ is produced in a fraction of a second after eight refinements.

$$(a_1, k_1, b_1, k_2) \rightarrow (a_2, 1 + k_2, b_1, k_2) \rightarrow (a_3, 1 + k_2, b_1, k_2) \rightarrow$$
$$(a_3, 1 + k_2, b_2, 2 + k_2) \rightarrow (a_3, 1 + k_2, b_3, 2 + k_2)$$

This counterexample actually represents a family of traces, since it is parameterized by the constants k_1 and k_2, with $k_1, k_2 \geq 0$, which have been introduced by the ICS decision procedures.

In the case of lazy theorem proving, the *offline* integration is particular expensive, since restarts implies the reconstruction of ICS logical contexts repetitively. Memoization of the decision procedure calls does not improve the situation significantly, since the assignments produced by Chaff in subsequent calls usually do not have long enough

	no explain			explain		
depth	time	conflicts	calls to ICS	time	conflicts	calls to ICS
5	0.03	24	162	0.01	7	71
6	0.08	48	348	0.01	7	83
7	0.19	96	744	0.02	7	94
8	0.98	420	3426	0.05	29	461
9	2.78	936	7936	0.19	70	1205
10	8.60	2008	17567	0.26	85	1543
15	-	-	-	4.07	530	13468

Table 2. Online lazy theorem proving.

common prefixes. This observation, however, might not be generalizable, since it depends on the specific, randomized heuristics of Chaff for choosing variable assignments.

In an *online* integration, choices for propositional variable assignments are synchronized with extending the logical context of the decision procedures with the corresponding atoms. Detection of inconsistencies in the logical context of the decision procedures triggers backtracking in the search for variable assignments. Furthermore, detected inconsistencies are propagated to the propositional search engine by adding the corresponding inconsistency clause (or, using an explanation function, a good overapproximation of the minimally inconsistent set of atoms in the logical context). Since state-of-the-art SAT solvers such as Chaff are missing the necessary API for realizing such an online integration, we developed a homegrown SAT solver which has most of the features of modern SAT solvers and integrated it with ICS. The results of using this online integration for the Bakery example can be found in Table 2 for two different configurations.[1] For each configuration, we list the total time (in seconds), the number of conflicts detected by ICS, and the total number of calls to ICS. Altogether, using an explanation facility clearly pays off in that the number of refinement iterations (conflicts) is reduced considerable.

7 Related Work

There has been much recent work in reducing the satisfiability problem of Boolean formulas over the theory of equality with uninterpreted function symbols to a SAT problem [5, 12, 23] using *eager* encodings of possible instances of equality axioms. In contrast, lazy theorem proving introduces the semantics of the formula constraints *on demand* by analyzing spurious counterexamples. Also, our procedure works uniformly for much richer sets of constraint theories. It would be interesting experimentally to compare the eager and the lazy approach, but benchmark suites (e.g. www.ece.cmu.edu/~mvelev) are currently only available as encodings of Boolean satisfiability problems.

In research that is most closely related to ours, Barrett, Dill, and Stump [2] describe an integration of Chaff with CVC by abstracting the Boolean constraint formula

[1] The differences in the number of conflicts compared to Table 1 are due to the different heuristics of the SAT solvers used.

to a propositional approximation, then incrementally refining the approximation based on diagnosing conflicts using theorem proving, and finally adding the appropriate conflict clause to the propositional approximation. This integration corresponds directly to an online integration in the lazy theorem paradigm. Their approach to generate good explanations is different from ours in that they extend CVC with a capability of abstract proofs for overapproximating minimal sets of inconsistencies. Also, optimizations based on *don't cares* are not considered in [2]. The experimental results in [2] coincide with ours in that they suggest that lazy theorem proving without explanations (there called the *naive* approach) and offline integration quickly become impractical. Using equivalence checking for pipelined microprocessors, speedups of several orders of magnitude over their earlier SVC system are obtained.

8 Conclusion

We developed a bounded model checking (BMC) procedure for infinite-state systems and linear temporal logic formulas with constraints based on a reduction to the satisfiability problem of Boolean constraint logic. This procedure is shown to be sound, and although incomplete in general, we establish completeness for invariant formulas. Since BMC problems are propositionally intensive, we propose a verification technique based on a *lazy* combination of a SAT solver with a constraint solver, which introduces only the portion of the semantics of constraints that is relevant for constructing a BMC counterexample.

We identified a number of concepts necessary for obtaining efficient implementations of lazy theorem proving. The first one is specialized to BMC for asynchronous systems in that we generate partial Boolean assignments based on the structure of program for restricting the search space of the SAT solver. Second, good approximations of minimal inconsistent sets of constraints at reasonable cost are essential. The proposed any-time algorithm uses a mixture of structural dependencies between constraints and a linear number of reruns of the decision procedure for refining overapproximations. Third, offline integration and restarting the SAT solver results in repetitive work for the decision procedures. Based on these observations we realized a lazy, online integration in which the construction of partial assignments in the Boolean domain is synchronized with the construction of a corresponding logical context for the constraint solver, and inconsistencies detected by the constraint solver are immediately propagated to the Boolean domain. First experimental results are very promising, and many standard engineering can be applied to significantly improve running times.

We barely scratched the surface of possible applications. Given the rich set of possible constraints, including constraints over uninterpreted function symbols, for example, our extended BMC methods seems to be suitable for model checking open systems, where environments are only partially specified. Also, it remains to be seen if BMC based on lazy theorem proving is a viable alternative to specialized model checking algorithms such as the ones for timed automata and extensions thereof for finding bugs, or even to AI planners dealing with resource constraints and domain-specific modeling.

Acknowledgements. We would like to thank the referees for their invaluable comments for improving this paper. S. Owre, J. Rushby, and N. Shankar provided many useful inputs.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *5th Symp. on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.
2. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, 2002. To be presented at CAV 2002.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zh. Symbolic model checking without BDDs. *LNCS*, 1579, 1999.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. *LNCS*, 1633:470–482, 1999.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. *LNCS*, 1855:154–169, 2000.
7. E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
8. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking in an industrial setting. *LNCS*, 2101:436–453, 2001.
9. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Symposium on Logic in Computer Science*, pages 51–60. IEEE, 2001.
10. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. *LNCS*, 2102:246–249, 2001.
11. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
12. A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. *LNCS*, 1427:244–255, 1998.
13. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 31(1):58–70, 2002.
15. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. *LNCS*, 2031:98–112, 2001.
17. M.O. M¨oller, H. Rueß, and M. Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(6), 2002.
18. O. M¨oller and H. Rueß. Solving bit-vector equations. *LNCS*, 1522:36–48, 1998.
19. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
20. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
21. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

22. David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
23. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. *LNCS*, 1633:455–469, 1999.
24. H. Rueß and N. Shankar. Deconstructing Shostak. In *16th Symposium on Logic in Computer Science (LICS 2001)*. IEEE Press, June 2001.
25. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. *LNCS*, 1579:178–192, 1999.
26. H. Sa¨ıdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101. IEEE Computer Society Press, 1999.
27. Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
28. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

Lemmas on Demand for Satisfiability Solvers *

Leonardo de Moura and Harald Rueß
({demoura, ruess}@csl.sri.com)
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Abstract. We investigate the combination of propositional SAT checkers with constraint solvers for domain-specific theories such as linear arithmetic, arrays, lists and the combination thereof. Our procedure realizes a lazy approach to satisfiability checking of propositional constraint formulas by iteratively refining Boolean formulas based on lemmas generated on demand by constraint solvers.

1. Introduction

Many search and optimization problems can effectively be solved using propositional reasoning techniques. Finiteness, however, is an inherent restriction of propositional encodings, and computational systems and environment models are usually expressed more succinctly in logics enriched with domain-specific constraints. Planning problems in AI, for example, may involve solving numeric resource constraints, and program analyses often require reasoning about constraints in the combination of datatypes such as integers, arrays, lists, or bitvectors.

Given a decidable constraint theory, we address the problem of constructing effective solutions to the satisfiability problem for propositional combinations of constraints. Of course, satisfiability solvers for propositional constraint formulas can easily be obtained from the combination of a propositional SAT solver with decision procedures simply by converting the problem into disjunctive normal form, but the resulting algorithm is usually prohibitively expensive. Alternatively, propositional search capabilities can be added to theorem provers, but it seems to be more effective to augment propositional SAT solvers with theorem proving capabilities.

Here we look at the specific combination of SAT solvers with constraint solvers, and we propose a method that we call *lemmas on demand*, which invokes the constraint solver *lazily* in order to efficiently prune out spurious counterexamples, namely, counterexamples that are generated by the SAT solver but discarded by the theorem prover by interpreting the propositional

* This research was supported by SRI International internal research and development, the DARPA NEST program through Contract F33615-01-C-1908 with AFRL, and the National Science Foundation under grants CCR-00-86096 and CCR-0082560.

atoms. For example, the SAT solver might yield the satisfying assignment $p, \neg q$, where the propositional variable p represents the atom $x = y$, and q represents $f(x) = f(y)$. A decision procedure can easily detect the inconsistency in this assignment. More importantly, it can be used to generate a set of conflicting assignments that can be used to construct a lemma that further constrains the search. In the above example, the lemma $\neg p \vee q$ can be added as a new clause in the input to the SAT solver. This process of refining Boolean formulas is similar in spirit to the refinement of abstractions based on the analysis of spurious counterexamples or failed proof attempts [26, 25, 6, 16, 8, 14, 18].

From a set of inconsistent constraints in a spurious counterexample we obtain an *explanation* as an over-approximation of the minimal, inconsistent subset of these constraints. The smaller the explanation that is generated from a spurious counterexample, the greater the pruning in the subsequent search. In this way, the computation of explanations accelerates the convergence of our procedure.

The paper is structured as follows. Section 2 includes some background material, whereas Section 3 describes the *lemmas on demand* approach and various refinements thereof. Initial experience with this technique is reported in Section 4. Finally, in Section 6 we draw conclusions.

2. Background

We use the familiar concepts and notations of propositional logic and constraint logic. The truth values $true, false$ are assigned to propositional variables. A literal is a propositional variable or its negation, a clause c is a disjunction of literals, and a *CNF* formula is a conjunction of clauses. There is a linear-time satisfiability-preserving transformation into CNF [22]. A propositional SAT solver ($\mathcal{B}$*-sat*) is, for our purposes, a computable function that receives a *CNF* formula and returns either a satisfying truth assignment or *unsatisfiable* if such an assignment does not exist.

A (conjunctive) constraint solver, say $\mathcal{C}$*-sat*, for a constraint theory $\mathcal{C}$, is a computable function that checks whether or not a set of constraints in a theory $\mathcal{C}$ is satisfiable. For instance, a linear programming system is a constraint solver for linear arithmetic.

Given a constraint theory $\mathcal{C}$, the set of *Boolean constraints* $\mathsf{Bool}(\mathcal{C})$ includes all constraints in $\mathcal{C}$ and it is closed under conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, and negation $\neg$. The notions of satisfiability, inconsistency, satisfying assignment, and satisfiability solver are lifted to the set of Boolean constraints in the usual way.

Formulas in $\mathsf{Bool}(\mathcal{C})$ can be translated into equisatisfiable Boolean formulas as long as the consistency of sets of constraints in $\mathcal{C}$ is decidable.

Translation schemes between propositional formulas and Boolean constraint formulas are needed. Given a formula φ such a correspondence is easily obtained by abstracting constraints in φ with (fresh) propositional variables. Let α be a function that maps constraints in $\mathcal{C}$ to propositional variables. This mapping induces a mapping from boolean constraint formulas to propositional formulas. For example, the formula $\varphi \equiv x_0 \geq 0 \wedge x_1 = x_0 + 1 \rightarrow x_1 \geq 1$ over linear arithmetic is mapped to $\alpha(\varphi) = p_1 \wedge p_2 \rightarrow p_3$, where $\alpha(x_0 \geq 0) \mapsto p_1$, $\alpha(x_1 = x_0 + 1) \mapsto p_2$, and $\alpha(x_1 \geq 1) \mapsto p_3$. Moreover, an assignment ν for propositional variables induces a set of constraints. Thus, let γ be the function that performs such mapping. For instance, the assignment $\nu = \{p_1 \mapsto \mathit{false}, p_2 \mapsto \mathit{true}, p_3 \mapsto \mathit{false}\}$ induces the set $\gamma(\nu) = \{x_0 < 0, x_1 = x_0 + 1, x_1 < 1\}$. Now, it is easy to see that a CNF formula φ in $\mathsf{Bool}(\mathcal{C})$ is equisatisfiable with the Boolean formula (in CNF)

$$\alpha(\varphi) \wedge \Big(\bigwedge_{\{l_1,\ldots,l_n\} \in I(\varphi)} (\neg l_1 \vee \ldots \vee \neg l_n) \Big)$$

where $I(\varphi)$ is the set of subsets $\{l_1, \ldots, l_n\}$ of literals l_i in $\alpha(\varphi)$ such that its "interpretation" $\{\gamma(l_1), \ldots, \gamma(l_n)\}$ is inconsistent in $\mathcal{C}$. Thus, every $\mathsf{Bool}(\mathcal{C})$ formula can be transformed into an equisatisfiable Boolean formula as long as there is a constraint solver for $\mathcal{C}$. On the other hand, the reduction seems to be infeasible, since an exponential number of $\mathcal{C}$-*inconsistency* checks is required in the worst case. It has been observed, however, that in many practical cases only small fragments of the set of $\mathcal{C}$-*inconsistencies* is needed. The main problem here is to identify small subsets of the set of all $\mathcal{C}$-*inconsistencies* which are sufficient to establish satisfiability of the Boolean constraint formula at hand.

3. Lemmas on Demand

We propose an algorithm based on the refinement of Boolean formulas with inconsistency lemmas that are generated on demand. We restrict ourselves to formulas in CNF, since most Boolean SAT solvers expect their input to be in this format.

The procedure $\mathbf{sat}(\varphi)$ in Figure 1 combines a Boolean SAT solver $\mathcal{B}$-*sat* and a domain-specific constraint solver $\mathcal{C}$-*sat*. $\mathcal{B}$-*sat* generates a candidate Boolean assignment for $\alpha(\varphi)$. If there is no such candidate, the algorithm terminates, since φ is clearly unsatisfiable. Otherwise the satisfiability solver $\mathcal{C}$-*sat* is used to check whether or not the Boolean assignment ν determines a valid assignment for φ. If the assignment is not valid, new propositional clauses (*inconsistency lemmas*) are added to the propositional formula at hand. The procedure *refine* is crucial in that it generates such new clauses. In order to guarantee soundness, all (interpretations of) clauses returned by

```
procedure sat(φ)
  p := α(φ);
  loop
    ν := B-sat(p);
    if ν = unsatisfiable then return unsatisfiable
    else if C-sat(γ(ν)) then return satisfiable
         else p := p ∪ refine(ν)
```

Figure 1. Lemmas on Demand for $\mathsf{Bool}(\mathcal{C})$.

refine are assumed to be implied by φ. In addition, the algorithm is *complete* if at least one clause returned by *refine* is not subsumed by clauses already in p. Alternatively, completeness can also be achieved by disabling infinite loops in which *refine* is only adding clauses subsumed by clauses already in p. We will return to a discussion about specific implementations of *refine* functions in Section 3.2.

3.1. CONSTRAINT THEORIES: EXAMPLES

One advantage of our approach is that it works uniformly for a large class of constraint theories, since the main requirement on these theories is the decidability of the conjunctive satisfiability problem. We review some of the more important constraints theories with polynomial satisfiability problem for the conjunction of a constraints. It follows that the satisfiability problem for the corresponding Boolean constraint theories are all NP-complete. In the following we assume as given a countably infinite set V of variables, and conjunctions of constraints are represented by finite sets.

3.1.1. *Equality for Constants.*

Satisfiability of conjunctive constraints C consisting of equalities $x = y$ and disequalities $x \neq y$ for variables x, y can be decided in linear time in the size of C. First, a graph is built, where the nodes are the variables and there is an edge between nodes x and y iff C contains the equality $x = y$. Now, C is satisfiable iff for all $u \neq v$ in C it is the case that u and v are not connected in this graph.

3.1.2. *Equality for Uninterpreted Functions.*

Terms are either variables or applications $f(t_1, \ldots, t_n)$, where f is a function symbol in some given signature of arity n. Satisfiability for a conjunction of equations and disequations over terms is decidable in $O(n \log(n))$ using congruence closure [11]. Satisfiability procedures for theories such as the

one for $cons$, car, and cdr can be obtained using congruence closure algorithms [20] by adding all relevant instances of universally quantified axiom schemes such as $x = car(cons(x, y)$. Similarly, using Ackermann's trick [1] or a variation thereof, one can transform Boolean constraints over equalities for uninterpreted terms to an equisatisfiable Boolean problem with equations over variables as literals by adding all possible instances of the congruence axiom and renaming uninterpreted subterms with variables. In the worst case, the number of such axioms is proportional to the square of the length of the given formula.

3.1.3. *Theories of Arithmetic.*
Linear arithmetic constraints are built up from inequalities over linear arithmetic terms including rational constants and addition. When interpreted over the rationals, the conjunctive satisfiability problem for linear arithmetic constraints is polynomial, since it is equivalent to the linear programming problem, which is known to be polynomial; when interpreting linear arithmetic terms over the integers, the problem becomes NP-complete over the integers. The conjunctive satisfiability problem for nonlinear arithmetic constraints, which include also multiplication, is still decidable when interpreted over the rationals, but becomes undecidable over the integers. Pratt observed that most inequalities in program verification are of the form $x - y \leq c$, where c is constant. Think of a conjunction C of these constraints representing a directed graph whose nodes are labelled with variables and there is an edge from x to y of weight c for each constraint $x - y \leq c$. Now, C is satisfiable iff there exists a negative-weight cycle in this graph. Using then Bellman-Ford algorithm, satisfiability of C is decided in time quadratic to the number of variables in C. Shostak's [28] loop residue algorithm for linear constraints $a * x + b * y \leq c$ reduces to Pratt's algorithm when applied to difference constraints.

3.1.4. *Theory of Fixed-Sized Bitvectors.*
A core theory of equalities over fixed-sized bitvectors includes variables x_n, which are interpreted over bitvectors of width n, extraction $x_n[i : j]$ of bits i through j, and concatenation of two bitvector terms. From the results in [7] it follows that the conjunctive satisfiability problem for this theory is decidable in polynomial time when the width of variables and extraction positions are integer constants. These problems can easily be translated to equisatisfiable propositional SAT problems by bitwise splitting of the bitvector constraints. In practice, however, considerable performance gains have been reported by using domain-specific bitvector procedures instead of SAT solvers [15]. For example, a bitvector encoding of the shift register BMC benchmark is exponentially more succinct than the corresponding Boolean formula [10].

3.1.5. *Combination of Satisfiability Procedures.*
Many verification problems require to solve constraint problems in the union of constraint theories. There are two basic paradigms for combining decision procedures. The Nelson-Oppen [21] method combines decision procedures for disjoint theories by exchanging equality information on the shared variables. If the constituent decision procedures are polynomial, then the combined Nelson-Oppen procedure is polynomial, too. In Shostak's method [29, 24, 27] the combination of the theory of pure equality with canonizable and solvable theories is decided through an extension of congruence closure that yields a canonizer for the combined theory. Again, if the constituent canonizers and solvers are polynomial-time, then Shostak's algorithm also runs in polynomial time. All of the individual theories listed above can be combined using either the Nelson-Oppen or the Shostak approach. Consequently, satisfiability for propositional logic with constraints in the combination of any subset of these theories is NP-complete.

3.2. REFINEMENTS

Now we describe some possible implementations of the *refine* function in Figure 1. A simple implementation of *refine* creates clauses of increasing size in each iteration. For example, if $\alpha(x_0 \geq 1) \mapsto p_1$, $\alpha(x_0 \geq 0) \mapsto p_2$, $\alpha(x_1 = x_0) \mapsto p_3$, $\alpha(x_1 \geq 1) \mapsto p_4$, $\alpha(x_1 \geq 0) \mapsto p_5$, the first call to *refine* produces the clauses $\neg p_1 \vee p_2$, and $\neg p_4 \vee p_5$, the second one produces the clauses $\neg p_1 \vee \neg p_3 \vee p_4$, $\neg p_1 \vee \neg p_3 \vee p_5$, and so on. This unguided enumeration is a sound and complete procedure, but it is usually infeasible in practice, since the number of clauses of size k is $O(n^k)$, where n is the number of constraints.

Alternatively, clauses are added in a guided way based on the analysis of the set of constraints corresponding to a Boolean assignment. For instance, if the Boolean assignment $\nu = \{p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_3 \mapsto \mathit{false}\}$ has been tested to yield an inconsistent set of constraints, the procedure *refine* adds the clause $\neg p_1 \vee p_2 \vee p_3$. This clause clearly prevents the invalid assignment to be regenerated by $\mathcal{B}$-*sat*. Therefore, the procedure of iteratively refining a Boolean formula based on the newly detected inconsistencies is terminating and complete. However, a naive implementation is also inefficient in practice, since only small fragments of the assignment ν are inconsistent. For example, suppose that an invalid assignment is associated with the following set of constraints:

$$\{x_0 \geq 0, y_0 \geq 0, x_1 = x_0, y_1 = y_0{+}1, x_2 = x_1{+}1, y_2 = y_1, x_2 \geq 1, x_2 < 1\}$$

It is clear that $\{x_2 \geq 1, x_2 < 1\}$ or $\{x_0 \geq 0, x_1 = x_0, x_2 = x_1 + 1, x_2 < 1\}$ are sufficient to describe the conflict. Therefore, let us assume that there is a function *explain* that returns an over-approximation of the minimal set

of constraints that implies the inconsistency detected by $\mathcal{C}$-*sat*. This function is similar to the conflict resolution procedures found in Boolean SAT solvers such as GRASP [17] or Chaff [19]. Abstractly, conflict resolution procedures in Boolean SAT solver can be seen as a function that receives a *conflicting clause* [1] and returns a new clause that prevents this specific conflict in future iterations. These new clauses are called *conflict clause*, and the process of constructing them is sometimes referred to as *learning*. There is an obvious trade-off between the conciseness of this approximation and the cost for computing it. We are proposing an algorithm for finding such an over-approximation based on rerunning the constraint solver $O(m \times n)$ times, where m is some given upper bound on the number of iterations (see below) and n is the number of given constraints.

The run in Figure 2 illustrates this procedure. The constraints in Figure 2.(a) are asserted to $\mathcal{C}$-*sat* from left-to-right. Since $\mathcal{C}$-*sat* detects a conflict when asserting $y_6 \leq 0$, this constraint is in the minimal inconsistent set. Now, an over-approximation of the minimal inconsistent sets is produced by connecting constraints with common variables (Figure 2.(a)). This over-approximation is iteratively refined by collecting the constraints in an array as illustrated in Figure 2.(b). Configurations consist of triples (C, l, h), where C is a set of constraints guaranteed to be in the minimal inconsistent set, and the integers l, h are the lower and upper bounds of constraint indices still under consideration. The initial configuration in our example is $(\{y_6 \leq 0\},\ 0,\ 3)$. In each refinement step we maintain the invariant that $C \cup \{array[i] \mid l \leq i \leq h\}$ is inconsistent. Given a configuration (C, l, h), individual constraints of index between l and h are added to C until an inconsistency is detected. In the first iteration of our running example we process constraints from right-to-left, and an inconsistency is only detected when processing $y_5 > 0$. The new configuration $(\{y_6 \leq 0, y_5 > 0\}, 1, 3)$ is obtained by adding this constraint to the set of constraints already known to be in a minimal inconsistent set, by leaving h unchanged, and by setting l to the increment of the index of the new constraint. The order in which constraints are asserted is inverted after each iteration. Thus, in the next step in our example, we successively add constraints between 1 and 3 from left-to-right to the set $\{y_6 \leq 0, y_5 > 0\}$. An inconsistency is first detected when asserting $y_6 = y_5$ to this set, and the new configuration is obtained as $(\{y_6 \leq 0, y_5 > 0, y_6 = y_5\}, 1, 1)$, since the lower bound l is now left unchanged and the upper bound is set to the decrement of the index of the constraint for which the inconsistency has been detected. The procedure terminates if C in the current configuration is inconsistent or after m refinements. In our example, two refinement steps yield the minimal inconsistent set $\{y_5 > 0, y_6 = y_5, y_6 \leq 0\}$. In general, the number of assertions is linear in the number of constraints, and the algorithm

[1] A conflicting clause is a clause in which all literals are assigned to *false*.

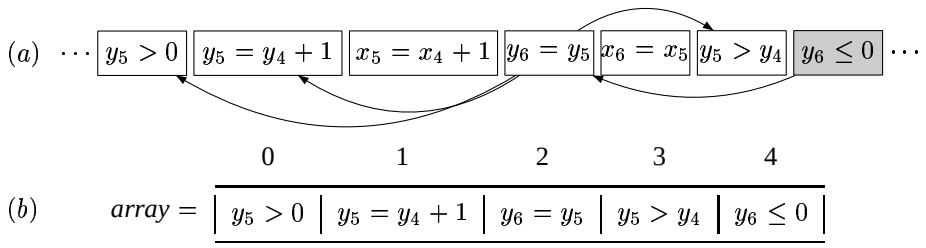

Figure 2. Trace for linear time *explain* function.

procedure *collect*(f, ν)
  if $f \equiv c_1 \vee \ldots \vee c_n$ **then**
    if $f \in \gamma(\nu)$ **then return** *collect*(*choose*($\{c_i \mid c_i \in \gamma(\nu)\}$, ν)
    else return $\bigcup_{i\in[1,n]}$ *collect*(c_i, ν)
  if $f \equiv c_1 \Leftrightarrow c_2$ **then**
    return *collect*$(c_1, \nu) \cup$ *collect*(c_2, ν)
  if $f \equiv \neg c$ **then**
    return *collect*(c, ν)
  if is-constraint(f) **then**
    if $f \in \gamma(\nu)$ **then return** $\{f\}$ **else return** $\{\neg f\}$
  return $\emptyset$

Figure 3. Collecting relevant constraints.

returns the exact minimal set if its cardinality is less than or equal to the upper bound m of iterations.

An additional refi nement can be introduced in the procedure $\mathbf{sat}(\varphi)$, since, in most cases, for a given assignment ν only a small subset of $\gamma(\nu)$ need to be considered. Overly eager assignments result in both useless search and overly specifi c counterexamples. For instance, assume the formula $(q \wedge p_1) \vee (\neg q \wedge p_2)$, and the assignment $\nu = \{q \mapsto \mathit{true}, p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{true}\}$, suppose the following two situations:

1. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x = -1$, $\mathcal{C}$-*sat*$(\gamma(\nu))$ returns unsatisfi able, since $\{x \geq 0,\ x = -1\}$ is inconsistent. Therefore the assignment ν is discarded and the search continues. However, constraint p_2 is clearly irrelevant, that is, it is a *don't care*.

2. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x \leq 1$, $\mathcal{C}$-*sat*$(\gamma(\nu))$ returns satisfi able. However, the resulting set of models is overly specifi c in that the value of x is restricted to those in the interval $[0, 1]$.

To solve this problem we keep the structure of the formula before CNF translation. The structure of formula is used to decide whether a constraint

procedure *refine-1*(ν) **return** { *clausify*($\gamma(\nu)$) }
procedure *refine-2*(ν) **return** { *clausify*(*explain*($\gamma(\nu)$)) }
procedure *refine-3*(ν) **return** { *clausify*(*collect*(φ, ν)) }
procedure *refine-4*(ν) **return** { *clausify*(*explain*(*collect*(φ, ν))) }
procedure *clausify*(C) **return** $\{\alpha(l) \mid \exists l' \in C \wedge l = \neg l'\}$

Figure 4. Refinement functions.

is relevant in a given assignment or not. The procedure $\mathbf{collect}(f, \nu)$ in Figure 3 collects all relevant constraints for a formula f and an assignment ν. For simplicity, this procedure only considers the propositional connectives: $\vee$, $\Leftrightarrow$, and $\neg$. The CNF translation adds a new propositional variable for each non-atomic sub-formula. It is important to notice that, $f \in \gamma(\nu)$ *iff* $\nu(\alpha(f)) = \mathit{true}$, that is, the formula f is assigned to *true* in the assignment ν. The function *is-constraint*(f) returns *true*, if f is a constraint. For instance, the formula $(q \wedge p_1) \vee (\neg q \wedge p_2)$ is represented as $\neg(\neg q \vee \neg p_1) \vee \neg(q \vee \neg p_2)$, and is translated to the following CNF formula:

$$
\begin{array}{l}
(a_1 \vee a_2) \wedge \\
(\neg q \vee \neg p_1 \vee a_1) \wedge (\neg a_1 \vee q) \wedge (\neg a_1 \vee p_1) \wedge \\
(q \vee \neg p_2 \vee a_2) \wedge (\neg a_2 \vee \neg q) \wedge (\neg a_2 \vee p_2)
\end{array}
$$

where, a_1 and a_2 are auxiliary propositional variables, that is, $a_1 \equiv \neg(\neg q \vee \neg p_1)$ and $a_2 \equiv \neg(q \vee \neg p_2)$. Given an assignment $\nu = \{q \mapsto \mathit{true}, p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{true}, a_1 \mapsto \mathit{true}, a_2 \mapsto \mathit{false}\}$, it is clear that $\mathbf{collect}(f, \nu) = \{q, p_1\}$, that is, the value of p_2 is a *don't care*.

Figure 4 summarizes the guided *refinement* procedures discussed above. The procedure *refine-1* implements the *naive* approach without explanation capability and no specific consideration of *don't cares*. The procedure *clausify* converts a set of conflicting constraints to a clause. Procedure *refine-2* uses the explanation facility but no *don't cares*, whereas *refine-3* uses explanations and handles *don't cares* by collecting relevant constraints with *collect* in Figure 3. Finally, the procedure *refine-4* uses all optimizations described in this section.

3.3. ONLINE INTEGRATION

So far, we described an *offline* integration of $\mathcal{B}$*-sat* and $\mathcal{C}$*-sat*, in which the solvers are treated as black boxes, and both procedures are restarted in each refinement step. However, some $\mathcal{C}$*-sat* tools support *backtracking*. In this case, an *online* integration is more appropriate, where choices for propositional variable assignments are synchronized with extending the logical context of the $\mathcal{C}$*-sat* with the corresponding atoms. Detection of inconsistencies in

```
procedure dpll()
  loop
    if decide() = done then return satisfiable
    loop
      cc := bcp();
      if cc = nil then break
      if not conflict-resolution(cc) then
        return unsatisfiable
```

Figure 5. Davis-Putnam procedure.

the logical context of the $\mathcal{C}$-*sat* triggers backtracking in the search for variable assignments. Furthermore, detected inconsistencies are propagated to the propositional search engine by adding the corresponding inconsistency clause (or, using an explanation function, a good over-approximation of the minimally inconsistent set of atoms in the logical context).

Figure 5 contains the main loop of the Davis-Putnam procedure found in most Boolean SAT solvers [17]. The algorithm starts with an empty boolean assignment, and traverses the space of truth assignments implicitly using a backtrack search algorithm. The search process iteratively performs the following steps: extends the current assignment by making a decision assignment to an unassigned variable (procedure *decide*); extends the current assignment by following logical consequences of the assignments made so far (procedure *bcp*), the deduction process may also identify and return a *conflicting clause* (variable *cc*), implying that the current assignment is not satisfiable; undoes (*backtracks*) the current assignment, if a conflict was detected, thus allowing another assignment to be tried (procedure *conflict-resolution*). The procedure *bcp* implements the boolean constraint propagation which corresponds to the application of the unit clause rule proposed by M. Davis and H. Putnam [9].

As described above, the *explain* function is similar to the conflict resolution procedure found in Boolean SAT solvers [17, 19]. Therefore, the conflict resolution procedure can be used to refine the result produced by the *explain* function in an online integration. For instance, suppose that the *explain* function returns the set $\{y > 10,\ y < 3\}$ as an explanation for a conflict detected by $\mathcal{C}$-*sat*. Then, this set is used to build the conflicting clause $\{\neg(y > 10),\ \neg(y < 3)\}$, which is sent to the conflict resolution procedure in $\mathcal{B}$-*sat*. A *conflict clause* is then produced by $\mathcal{B}$-*sat*. Figure 6 contains our *online* algorithm. The procedure *propagate-to-*$\mathcal{C}$ is responsible to send recently assigned constraints to $\mathcal{C}$-*sat*, it returns a conflicting clause if an inconsistency is detected. In other words, the procedure *propagate-to-*

```
procedure C-dpll()
  loop
    status := decide();
    if status = done or should-propagate-to–C then
      cc := propagate-to-C();
      if cc ≠ nil then
        status := not-done;
        if not conflict-resolution(cc) then
          return unsatisfiable
    if status = done then return satisfiable
    loop
      cc := bcp();
      if cc = nil then break
      if not conflict-resolution(cc) then
        return unsatisfiable
procedure propagate-to-C()
  relevant-constrains := collect(φ, ν);
  if C-assert(relevant-constrains) return nil
  return clausify(explain(relevant-constrains))
```

Figure 6. Online Integration.

$\mathcal{C}$ implements the *bridge* between $\mathcal{B}$-*sat* and $\mathcal{C}$-*sat*. In our *online* algorithm, the procedure *collect* behaves slightly different, since it must handle unassigned variables, since ν can be a partial assignment. We also keep track of which constraints were already sent to $\mathcal{C}$-*sat*, so the procedure *collect* only collects the *unsent constraints*. The procedure $\mathcal{C}$-*assert* is an incremental version of procedure $\mathcal{C}$-*sat* in Figure 1, that is, it extends the logical context of $\mathcal{C}$-*sat* with the new constraints in the variable *relevant-constrains*. The procedure *propagate-to-*$\mathcal{C}$ is called when a satisfiable boolean assignment is found (*decide* returns *done*), or when the flag *should-propagate-to-*$\mathcal{C}$ is *active*. Different heuristics can be used to activate this flag, in our implementation it is activated every time a given number of new constraints are assigned to a boolean value. So, if the problem only contains propositional variables, our algorithm will behave like a standard Boolean SAT solver. Although it is not described in the Figure 6, the procedure *decide* must request $\mathcal{C}$-*sat* to create a new *backtracking point*, and *conflict-resolution* must request $\mathcal{C}$-*sat* to execute the *backtracking*. Our integrated algorithm is compatible with any kind of decision heuristic and standard optimizations such as *non-chronological backtracking* and *learning* [17].

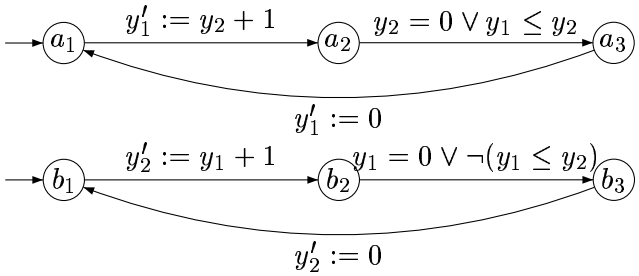

Figure 7. Bakery Mutual Exclusion Protocol.

4. Experiments

We implemented several refi nements of the basic lazy theorem proving algorithm from Section 3, using Chaff [19] for the *offline* integration, and ICS [12] for deciding constraints. ICS is a ground decision procedure for the combination of linear arithmetic constraints, the theory of tuples, arrays, bitvectors, and equality over uninterpreted functions. Since state-of-the-art Boolean SAT solvers such as Chaff are missing the necessary API for realizing such an online integration, we used a home-grown SAT for realizing the online integration. We describe some of our experiments using the Bakery mutual exclusion protocol in Figure 7 [2] with initial states $y_1 \geq 0 \wedge y_2 \geq 0$. The basic idea is that of a bakery, where customers take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section. In our example, there are only two processes (P_1 and P_2). The program location $a_3(b_3)$ represents the critical section of the process $P_1(P_2)$. The variable $y_1(y_2)$ contains the number that $P_1(P_2)$ uses to enter the critical section, it is zero if the process is not trying to enter the critical section. Only one process can execute a transition at each time. In this example, we are interested in the property that the processes are never in their critical sections at the same time. For validating this property we use bounded model checking (BMC) to search for counterexamples of length k to the model checking problem $M \models \varphi$, where M is the system (program) being verifi ed, and φ is the mutual exclusion property. This technique has been introduced for fi nite systems in [4]. Here, we are working with an extension of the BMC methodology to infi nite-state systems [10, 30].

We use the convention that current variables are always written as y_1, y_2 whereas the next-state variables are written as y_1', y_2'. In addition, x^i represents the value of the variable x at time i. The variable $pc_1(pc_2)$ is the *program counter* of the process $P_1(P_2)$. Thus the formula that describes the initial state is:

$$pc_1^0 = a_1 \wedge y_1^0 \geq 0 \wedge pc_2^0 = b_1 \wedge y_2^0 \geq 0$$

[2] See also http://www.csl.sri.com/~demoura/bmc-examples.

We want to verify the property $\neg(pc_1 = a_3 \wedge pc_2 = b_3)$, thus, a counterexample of length k is a trace that reaches the *goal* ($pc_1^k = a_3 \wedge pc_2^k = b_3$). The transitions are encoded as:

$$
\begin{aligned}
&(pc_1^i = a_1 \wedge y_1^{i+1} = y_2^i + 1 \wedge pc_1^{i+1} = a_2 \wedge pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \vee \\
&(pc_1^i = a_2 \wedge (y_2^i = 0 \vee y_1^i \leq y_2^i) \wedge y_1^{i+1} = y_1^i \wedge pc_1^{i+1} = a_3 \wedge \\
&\quad pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \vee \\
&(pc_1^i = a_3 \wedge y_1^{i+1} = 0 \wedge pc_1^{i+1} = a_1 \wedge pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \vee \\
&(pc_2^i = b_1 \wedge y_2^{i+1} = y_1^i + 1 \wedge pc_2^{i+1} = b_2 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i) \vee \\
&(pc_2^i = b_2 \wedge (y_1^i = 0 \vee \neg(y_1^i \leq y_2^i)) \wedge y_2^{i+1} = y_2^i \wedge pc_2^{i+1} = b_3 \wedge \\
&\quad pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i) \vee \\
&(pc_2^i = b_3 \wedge y_2^{i+1} = 0 \wedge pc_2^{i+1} = b_1 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i)
\end{aligned}
$$

This enconding includes the *frame axioms* to describe which variables a transition does *not* affect. The program counter (pc_1 and pc_2) can be encoded using propositional variables, since their domains are finite.

Table I. Offline lazy theorem proving ('-' is time ≥ 1800 secs).

depth	*refine-2*		*refine-3*		*refine-4*	
	time	conflicts	time	conflicts	time	conflicts
5	45.23	577	0.71	66	0.31	16
6	83.32	855	2.36	132	0.32	18
7	286.81	1405	12.03	340	1.75	58
8	627.90	1942	56.65	710	2.90	73
9	1321.57	2566	230.88	1297	8.00	105
10	-	-	985.12	2296	15.28	185
15	-	-	-	-	511.12	646

Table I includes some statistics for three different *offline* configurations depending on which *refine* procedure described in Section 3 is used. For each configuration, we list the total time (in seconds) and the number of conflicts detected by the decision procedure. This table indicates that the effort of detecting the relevant constraints, and the linear *explain* function are essential for efficiency. Recall that the experiments so far represent worst-case scenarios in that the given formulas are unsatisfiable. For BMC problems with counterexamples, however, our procedure usually converges much faster. Consider, for example the mutual exclusion problem of the Bakery protocol with the assignment $y_2' := y_2 + 1$ instead of $y_2' := y_1 + 1$. The corresponding counterexample for $k = 7$ is produced in a fraction of a second after adding 53 lemmas.

Table II. Online lazy theorem proving.

depth	no explain			explain		
	time	conflicts	calls to ICS	time	conflicts	calls to ICS
5	0.03	24	162	0.01	7	71
6	0.08	48	348	0.01	7	83
7	0.19	96	744	0.02	7	94
8	0.98	420	3426	0.05	29	461
9	2.78	936	7936	0.19	70	1205
10	8.60	2008	17567	0.26	85	1543
15	-	-	-	4.07	530	13468

$$
\begin{array}{lll}
(a_1, k_1, b_1, k_2) & \rightarrow (a_1, k_1, b_2, 1 + k_2) & \rightarrow \\
(a_2, 2 + k_2, b_2, 1 + k_2) & \rightarrow (a_2, 2 + k_2, b_3, 1 + k_2) & \rightarrow \\
(a_2, 2 + k_2, b_1, 0) & \rightarrow (a_3, 2 + k_2, b_1, 0) & \rightarrow \\
(a_3, 2 + k_2, b_2, 1) & \rightarrow (a_3, 2 + k_2, b_3, 1) &
\end{array}
$$

Notice that this counterexample represents a family of traces, since it is parametrized by (newly introduced constants) k_1 and k_2 with $k_1, k_2 \geq 0$.

The results of using this *online* integration for the Bakery example can be found in Table II for two different configurations.[3] For each configuration, we list the total time (in seconds), the number of conflicts detected by ICS, and the total number of calls to ICS. Altogether, using an explanation facility clearly pays off in that the number of refinement iterations (conflicts) is reduced considerable.

5. Related Work

For the special case of equality theories over terms with uninterpreted function symbols, Ackermann [1] already defined a reduction to Boolean logic by adding propositional encodings of all relevant instances of the congruence axiom. Variations of Ackermann's trick have been used, for example, by Shostak [28] for arithmetic reasoning in the presence of uninterpreted function symbols, and various reductions of the satisfiability problem of Boolean formulas over the theory of equality with uninterpreted function symbols to propositional SAT problems have recently been described by Goel, Sajid, Zhou, and Aziz [13], by Pnueli, Rodeh, Shtrichman, and Siegel [23], and by

[3] The differences in the number of conflicts compared to Table I are due to the different heuristics of the SAT solvers used.

Bryant, German, and Velev [5]. In a similar vein, an eager reduction to propositional logic for constraints in Pratt's difference logic have been described by Strichman, Seshia, and Bryant [31]. Even for such a simple constraint theory, however, an exponential number of constraints may be generated in the preprocessing stage.

Compared with these *eager* reductions, our *lazy* integration procedure uniformly works for logics with a rich set of data types. Moreover, instead of constructing an equisatisfiable Boolean formula *a priori*, we compute a sequence of refinements by adding propositional lemmas as obtained from an analysis of spurious propositional assignments. In this way, the semantics of constraints is introduced gradually and on *on demand*. In this way, only inconsistency lemmas of relevance to the satisfiability of the formula are added.

In research that is most closely related to ours, Barrett, Dill, and Stump [3] describe an integration of Chaff with CVC by abstracting the Boolean constraint formula to a propositional approximation, then incrementally refining the approximation based on diagnosing conflicts using theorem proving, and finally adding the appropriate conflict clause to the propositional approximation. This integration corresponds directly to an online integration in the *lemmas on demand* paradigm. Their approach to generate good explanations is different from ours in that they extend CVC with a capability of abstract proofs for over-approximating minimal sets of inconsistencies. Also, optimizations based on *don't cares* are not considered explicitly in [3]. The experimental results in [3] coincide with ours in that they suggest that lazy theorem proving without explanations (there called the *naive* approach) and offline integration quickly become impractical. Using equivalence checking for pipelined microprocessors, speedups of several orders of magnitude over their earlier SVC system are obtained.

Armando, Castellini, and Giunchiglia [2] propose a SAT-based approach for the special case of solving disjunctions of Pratt's difference constraints. In their experiments, they observe excessively redundant computations, which can largely be eliminated using our explanation capabilities. A preprocessing step for computing inconsistency clauses with two literals is used [2] to simplify problems. We also found it to often to be advantageous to pregenerate 2- and even 3-inconsistencies to accelerate convergence. Optimizations based on *don't cares* are not considered in [2].

6. Conclusion

The main contribution of this paper is a *lazy* integration of propositional SAT solvers with constraint solvers for effectively deciding the satisfiability problem for propositional constraint formulas. The key idea is to use constraint

solvers for suggesting, on demand, useful inconsistency lemmas. In this way, only inconsistency lemmas of relevance to the satisfiability of the formula are added. Various refinements such as online integration and acceleration of convergence using explanation functions are needed to make the *lemmas on demand* approach work effectively in practice.

References

1. W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.
2. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the Fifth European Conference on Planning (ECP-99)*, 1999.
3. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, 2002. To be presented at CAV 2002.
4. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, 1999.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 154–169, Chicago, IL, 2000. Springer-Verlag.
7. David Cyrluk, Harald Rueß, and Oliver Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer-Aided Verification, CAV '97*, volume 1254 of *LNCS*, pages 60–71, Haifa, Israel, 1997. Springer-Verlag.
8. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Symposium on Logic in Computer Science*, pages 51–60. IEEE, 2001.
9. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
10. Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer-Verlag.
11. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.
12. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proceedings of CAV'2001*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.
13. A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 244–255. Springer-Verlag, 1998.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 31(1):58–70, 2002.
15. P. Johannsen and R. Drechsler. Formal verification on register transfer level—utilizing high-level information for hardware verification. In *IFIP International Conference on Very Large Scale Integration (VLSI'01)*, pages 127–132, Montpellier, France, 2001.

16. Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.
17. Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD'96*, pages 220–227, 1996.
18. M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(2), 2002.
19. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, 2001.
20. G. Nelson and D. Oppen. Fast decision algorithms based on congruence closure. Technical Report STAN-CS-77-646, Computer Science Department, Stanford University, 1977.
21. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
22. David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.
23. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 455–469, Trento, Italy, 1999. Springer-Verlag.
24. Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.
25. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, March 1999. Springer-Verlag.
26. H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101. IEEE Computer Society Press, 1999.
27. N. Shankar and Harald Rueß. Combining shostak theories. Invited paper for Floc'02/RTA'02, 2002.
28. Robert E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
29. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
30. Maria Sorea. Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science*, 68(5), 2002.
31. O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding Separation Formulas with SAT. In *International Conference on Computer-Aided Verifi cation (CAV'02)*. Springer Verlag, July 2002.

Presented at the National Security Agency's third High Confidence Software and Systems Conference, Baltimore MD, April 2003

Embedded Deduction With ICS*

Leonardo de Moura, Harald Rueß, John Rushby, and Natarajan Shankar

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
demoura|ruess|rushby|shankar@csl.sri.com

Abstract. Formal analyses can provide valuable assurance for high confidence software and systems. The analyses can range from strong typechecking through test case generation and static analysis to model checking and full verification. In all cases, the tools that support the analyses use formal deduction in some way or other. ICS is a fully automatic, high-performance decision procedure for a broad combination of theories that can be embedded in all tools of this kind to provide them with a core deductive capability of exceptional power and performance. We describe the design choices underlying ICS and the capabilities it provides.

1 Introduction

Formal deduction—that is, automated theorem proving—lies at the heart of all tools for formal analysis of software and system descriptions. In formal verification systems such as PVS [10], the deductive capability is explicit and visible to the user, whereas in tools such as test case generators it is hidden and often ad-hoc. We believe that all tools for formal analysis would benefit—both in performance and ease of construction—if they could draw on a powerful embedded service to perform common deductive tasks.

Examples of the tasks that can be required are those that ask whether one formula is a consequence of others (e.g., is $4 \times x = 2$ a consequence of $x \leq y$, $x \leq 1 - y$, and $2 \times x \geq 1$ when the variables range over the reals?), and those that ask whether an assignment to variables can be found that satisfies a set of constraints (e.g., find an a such that $car(a) = cons(b, c)$). The first task is a decision problem that might arise in verification, the second is a constraint satisfaction problem that could arise in test case generation. Notice that both examples involve interpreted theories: rational linear arithmetic in the first, and lists in the second.

An embedded deductive service should be fully automatic, and this suggests that its focus should be restricted to those theories whose decision and satisfiability problems are decidable. However, there are some contexts that can tolerate incompleteness (e.g., in extended static checking, the failure to prove a true theorem results only in a spurious warning message), and others where speed may be favored over completeness (e.g., in

* This research was supported by SRI internal investment funds, by NASA under contract NAS1-00079, by the DARPA NEST program under AFRL contract F33615-01-C-1908, and by NSA under contract MDA904-02-C-1196

construction of abstractions), so that undecidable theories (e.g., nonlinear integer arithmetic) and those whose decision problems are often considered infeasible in practice (e.g., real closed fields) should not be ruled out completely.

Most problems that arise in practice involve *combinations* of theories: the question whether

$$f(cons(4 \times car(x) - 2 \times f(cdr(x)), y)) = f(cons(6 \times cdr(x), y))$$

follows from $2 \times car(x) - 3 \times cdr(x) = f(cdr(x))$, for example, requires simultaneously the theories of uninterpreted functions, linear arithmetic, and lists. The ground (i.e., quantifier-free) fragment of many combinations is decidable when the full (i.e., quantified) combination is not, and practical experience indicates that automation of the ground case is adequate for most applications.

Practical experience also suggests several other desiderata for an effective deductive service. Some applications (e.g., construction of abstractions) invoke their deductive service a huge number of times in the course of a single calculation, so that performance of the service must be very good (e.g., tens or hundreds of thousands of invocations per second). Other applications (e.g., proof search) explore many variations on a formula (i.e., alternately asserting and denying various combinations of its premises), so the deductive service should not examine individual formulas in isolation, but should provide a rich API that supports incremental assertion, retraction, and querying of formulas. Other applications (e.g., test case generation) generate propositionally complex formulas (i.e., formulas with thousands or millions of propositional connectives applied to terms over the decided theories), so that this type of proof search must be performed efficiently inside the deductive service.

We have developed a system called ICS (the name stands for *Integrated Canonizer/Solver*) that can be embedded in applications to provide deductive services satisfying the desiderata above. In the following sections, we outline the design choices embodied in ICS, its capabilities and method of operation, and describe some of its applications.

2 Core ICS

The core of ICS is a decision procedure for a combination of ground theories including equality with function symbols, integer and rational linear arithmetic, fixed-length bitvectors, arrays, tuples, and coproducts (the combination of the last two provides abstract datatypes such as lists and binary trees). Apart from bitvectors, this capability is similar to that of the decision procedures in PVS (e.g., the `assert` command), but ICS can handle much larger formulas.

It is crucial to its utility that ICS is able to decide a *combination* of theories. It is desirable to achieve this by combining decision procedures for its individual theories in a modular fashion. However, there is a tradeoff between modularity and performance. The combination method of Nelson and Oppen [9], for example, imposes few restrictions on its component theories and their decision procedures, but yields relatively low performance. This is because the separate decision procedures do not share much state and communicate only by propagating newly discovered equalities back and forth. The

combination method of Shostak [14], on the other hand, requires that its component theories are *canonizable* and *solvable*, and achieves high performance by tightly integrating these components through an efficient data structure for congruence closure. Most theories of practical interest are canonizable and solvable, so ICS uses a corrected version of Shostak's method. Theories that do not satisfy the requirements for Shostak's method can be integrated using Nelson and Oppen's method above the Shostak combination.

As mentioned, an efficient data structure and procedure for congruence closure lies at the heart of ICS. This provides a decision procedure for the theory of equality with uninterpreted function symbols, and is used to integrate decision procedures for other canonizable and solvable theories. Early treatments of this integration were incorrect and could yield incomplete or nonterminating procedures. The first correct treatment for the integration of congruence closure with one other theory was developed by Shankar and Rueß [12]; this construction has been formally verified in PVS by Ford and Shankar [6]. The extension to multiple theories is not straightforward because, although the combination of the canonizers for the constituent theories yields a canonizer for the combined theory (which is an independently useful artifact), the combination of the solvers may not (contrary to previous belief) be a solver for the combination. The first correct extension to multiple theories also was developed by Shankar and Rueß [13].

A decision procedure (i.e., canonizer and solver) for rational linear arithmetic is quite straightforward and efficient, but integer linear arithmetic is more challenging because it can require case-splitting (i.e., search) to determine whether some property is satisfied by an integer in a certain range (hence, the problem is NP-complete). There are straightforward methods for this problem that are easily shown to be complete (e.g., the method of Fourier-Motzkin), but they are inefficient on cases that commonly arise in practice (e.g., constraints of the form $x - y \leq c$, where x, y are variables and c is an integer constant). ICS uses a new method that is efficient on the common cases, complete, and smoothly extensible to richer fragments such as nonlinear arithmetic.

Verification and model checking for hardware generally involve reasoning over bitvectors. It is, of course, possible to treat each bit as a Boolean variable and then use an efficient decision procedure for the Booleans, but this immediately invites an exponential case explosion. A better method is to split the bitvectors into chunks (not individual bits) and to do so only when necessary. ICS uses a method of this kind for fixed-length bitvectors [2, 7] and integrates it with integer arithmetic for their numerical (e.g., unsigned and twos-complement) interpretations.

In addition to the theories described above, ICS also decides the theories of arrays, tuples, and coproducts; the combination of the latter two can represent abstract datatypes such as lists and binary trees.

Core ICS operates as a decision procedure: it reports whether the formula under consideration is valid—which is equivalent to its negation being unsatisfiable. In the case that a formula is satisfiable, the ICS data structures contain sufficient information to extract a satisfying assignment—although this is not yet implemented.

3 ICS with SAT

Core ICS operates on formulas that are conjunctions of terms in the combination of its theories. However, many applications generate proof obligations or constraints that have richer propositional structure. For example, a test case of length 2 for a shift register may reduce to satisfiability of the following formula.

$$(x_1 = x_0[1:n-1] +\!+ 1_1) \wedge (x_2 = x_1[1:n-1] +\!+ 1_1) \wedge$$
$$(x_0 \neq 0_n \vee x_1 \neq 0_n \vee x_2 \neq 0_n) \wedge (x_0 = x_2 \vee x_1 = x_2).$$

where $x[1:r]$ denotes extraction of bits 1 through r of the bitvector x of length n, $+\!+$ denotes bitvector concatenation, and 1_r (resp. 0_r) denotes the bitvector of length r whose bits are all 1 (resp. 0).

The disjunctions in formulas such as this necessitate search and the challenge is to integrate this capability with core ICS. The PVS `ground` command provides modest functionality of this type with the assistance of an external BDD package. The problem with this approach is that the BDD represents all possible satisfying assignments (and is therefore expensive to construct), whereas we would be satisfied with just one (or the knowledge that there are none). Propositional satisfiability solvers (SAT solvers) provide this more targeted type of search and recent advances have made them extraordinarily fast for many problems that arise in practice—often they are able to discharge formulas with hundreds of thousands of variables and millions of terms in seconds or a few minutes [8].

To connect core ICS to a SAT solver, we use *variable abstraction*: each interpreted term (e.g., $x_1 = x_0[1:n-1] +\!+ 1_1$) is replaced by a distinct propositional variable (e.g., p) and the SAT solver is asked to solve the resulting propositional system. The truth values assigned to the propositional variables by the SAT solver are then extended to their original interpretations and the core ICS decision procedure checks them for consistency. If the interpretations are consistent, then we are done; if not, the root of the inconsistency can be generated and passed to the SAT solver as an additional constraint (we call this the generation of "lemmas on demand" [3]). For example, if p represents the term $x = y$, q represents $f(x) = f(y)$, and the SAT solver returns $p, \neg q$, then core ICS will detect the inconsistency in the interpretation $x = y \wedge f(x) \neq f(y)$ and can generate the lemma $\neg p \vee q$ as a new constraint for the SAT solver. Proceeding back and forth in this way, the SAT solver generates new candidate assignments and the decision procedure generates new additional constraints until either we find an assignment whose interpretation is satisfiable, or the set of constraints becomes unsatisfiable. The effectiveness of this approach depends on how rapidly the search space is cut down at each stage by the new constraints generated by the decision procedure. The most potent constraints would be the true "root causes" of the inconsistencies detected at each stage but it can take a long time to calculate such precise constraints and this negates the savings due to the smaller search space. Good overall performance is obtained using fast heuristics that generate an approximate "explanation" for the root cause of each inconsistency [3]. We are still tuning our heuristics in search of the best overall performance.

Full ICS integrates the combined decision procedure of core ICS with a SAT solver in the manner described. We do not use an off-the-shelf SAT solver because the back-and-forth interaction with the decision procedure imposes novel requirements (e.g., we

want to process new constraints incrementally from the current state, not restart from the beginning, and we also use "don't care" assignments), but we do employ many of the techniques that make such solvers fast [15]. Our experiments indicate that the integrated SAT solver in ICS yields several orders of magnitude improvement over a looser combination using an off-the-shelf SAT solver. Used purely as a SAT solver, the performance of full ICS is comparable to Chaff [8].

Like core ICS, full ICS operates as a decision procedure, but we plan to extend it to a satisfiability procedure in the near future.

4 Using ICS

Core ICS is implemented in Objective Caml, and its SAT solver in C++; the full system functions as a C library and can be called from virtually any language. We have experience using it from C, C++, Lisp, Scheme, and Objective Caml. The system was developed under Linux but has been ported to MAC OS X and to Windows XP (under cygwin), and we anticipate little difficulty in porting it to other systems.

In addition to its C interface, ICS is provided with a simple text-based interactor that can be used for experimenting with its capabilities. ICS maintains a state that can be manipulated and queried by a series of commands. Most importantly, the `assert` command extends the current state with a new fact. The following command, for example, adds an equality over terms built from the the variable `x`, the uninterpreted function symbol `f`, the operators of linear arithmetic, and S-expressions built from the pairing function `cons(.,.)` and its first and second projections `car(.)` and `cdr(.)`.

```
ics> reset.
:ok
ics> assert 2 * car(x) - 3 * cdr(x) = f(cdr(x)).
:ok
```

We can now assert a second equality, and the response `valid` indicates that this is deduced to be a consequence of the previously asserted facts.

```
ics> assert f(cons(4 * car(x) - 2 * f(cdr(x)), y))
       = f(cons(6 * cdr(x), y)).
:valid
```

The command `sat` invokes the SAT solver (here `|` denotes disjunction and `&` is conjunction).

```
ics> sat (x = 1 | x = 2 | x = 3) & x > 1.
:sat(s5) [-1 + x > 0;  x = 3]
```

The response from ICS indicates that all assignments to `x` satisfying both `-1 + x > 0` and `x = 3`, describe models for the input formula (the annotation `s5` simply

names this logical state). There is obviously only one possible assignment here, so the description is not minimal. Construction of concrete satisfying assignments is planned for the near future.

5 Applications of ICS

ICS can be used to provide embedded deductive support for existing applications, but its speed and power also make new applications possible. We describe representatives applications of each kind.

5.1 Discharging Proof Obligations

ICS can be used to augment or replace existing deductive capabilities in systems that generate and discharge proof obligations.

For example, ICS can be used in place of the standard decision procedures in PVS. Because the standard decision procedures have different capabilities than ICS, a PVS proof script developed using the former will generally require adjustment to work with the latter. For testing and benchmarking purposes, we have run PVS in a mode where proof scripts are guided by the standard decision procedures, but ICS is run in parallel and its behavior compared with the standard procedures. Differences were examined to ensure they were intended. We used proofs of the 750 theorems in the PVS prelude (built-in library) as our test bench. Despite its more costly interface (PVS and its standard decision procedures are implemented in Lisp, from which ICS is invoked as a foreign-function through its C interface) and the fact that PVS uses only its core capabilities, ICS is substantially faster on examples that really exercise the decision procedures (for small examples, any differences are swamped by the overhead of other processing in the PVS prover). Future versions of PVS will make fuller use of ICS capabilities. We anticipate that this will be beneficial both to users of PVS and to those who intend to use ICS directly but wish to use PVS to explore and prototype the deductive "glue" needed to reduce their application to the capabilities provided by ICS. Such glue is likely to involve Skolemization (and possibly quantifier instantiation), and definition expansion (and possibly rewriting).

We are currently optimizing the capabilities of ICS to support the deductive requirements of the Destiny verification system under development at NSA.

5.2 Bounded Model Checking and Test Case Generation

Bounded model checking (BMC) has become a popular debugging and assurance method for hardware designs [1]. Bounded model checking asks whether there is a counterexample of length k or less to a given property P (typically an invariant, but the method works for full linear temporal logic) of a design represented as an initiality predicate I and transition relation T. For hardware designs at the register transfer level, P, I, and T are represented directly in propositional calculus and the BMC problem then reduces to a (typically, huge) SAT problem. The performance of modern SAT solvers allows BMC to find deeper bugs on bigger designs than a standard BDD-based symbolic

model checker. More importantly, BMC requires less tinkering (e.g., variable ordering, downscaling) by the user than standard model checking. Typically, the process is to try $k = 1$, then $k = 2, 3, \ldots$ until either a counterexample is found, or the resources of the computer—or the patience of the user—are exhausted.

Full ICS immediately allows BMC to be extended from hardware designs consisting of purely Boolean circuits to software and system designs (and hardware designs at higher levels of description) whose state is defined over integers, arrays, bitvectors, and datatypes, and their corresponding operations—in short, over any combination of the theories decided by ICS. We call this "Infinite BMC" since the state space is potentially infinite [5].

Given a system specified by initiality predicate I and transition relation T, there is a counterexample of length k to invariant P if there is a sequence of states $s_0, \ldots, s_k$ such that

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k).$$

The Infinite BMC problem is simply to find a satisfying assignment for $s_0, \ldots, s_k$ in this formula—which is exactly the capability of ICS. [1]

Using correct designs supplied for evaluation purposes by an industrial collaborator (they are hardware designs, but we do not know their origins or purpose), we performed Infinite BMC for increasing k until the time taken by ICS approached 30 minutes (on a 2GHz Pentium IV with 1GB of memory). At this point, one of the BMC formulas had 227,108 terms and its representation as a text file occupied 5Mb, another had 105,844 terms and a 3Mb text file, while a third had 72,291 terms and a 2Mb text file. In all cases, ICS required less than 80 Mb of memory. Observe that these are worst-case examples: the designs are correct (for the invariants concerned) and hence the BMC formulas have no satisfying assignments and the full search space must be explored. Other invariants do manifest bugs in the second of the designs mentioned above, and ICS found a counterexample to one of them of length 4, and a counterexample to another of length 6, both in under a minute.

Structural test coverage criteria, including the MC/DC criterion required for flight control software, can be specified as formulas in temporal logic [11]. Counterexamples to the negation of these formulas then constitute suitable test cases. Experiments with symbolic model checkers have shown that they can be used within this framework as very effective test case generators. Bounded model checkers should be even more effective (since they are specialized to the efficient construction of counterexamples). However, these strictly Boolean and propositional methods apply only to Boolean abstractions of software designs specified over arithmetic variables and data structures and can therefore generate infeasible test cases. Infinite BMC using ICS can be applied directly to software designs, thereby eliminating infeasible test cases and achieving accurate coverage.

[1] As noted earlier, ICS currently operates as a decision procedure: it can indicate whether a formula is valid or, equivalently, whether its negation is unsatisfiable. In the case that the negation to a formula is satisfiable, ICS does not yet produce a satisfying assignment (i.e., a concrete counterexample to the original formula). However, the Infinite BMC procedure does extract "symbolic counterexamples" from information in the ICS data structures.

5.3 k-Induction

If BMC finds a counterexample of length k, then we have found a bug, and are done. But if we fail to find a counterexample for any k up to some limit on our resources or patience, we cannot conclude that we have verified the design—for there could always be a counterexample of length longer than any that we tried.[2] To verify the design (for safety property P), we must perform some kind of inductive argument that applies to traces of all lengths. The usual way to do this by theorem proving is to establish that the property concerned is *inductive*: that is, it is true of all initial states (i.e., $I(s) \supset P(s)$) and if it is true of some state, then it is true of all its successors (i.e., $P(s) \wedge T(s,t) \supset P(t)$). The weakness of this method is that the second condition may be violated by a state s that is unreachable from an initial state. We must then replace P by a stronger property that excludes the troublesome state s and repeat the process. It is not uncommon to have to iterate this process many tens of times. Strengthening often requires human insight, though a good heuristic is often to conjoin to P a formula that asserts that s is unreachable.

A stronger form of induction requires that only when we have a sequence of k states satisfying P must all the successors also satisfy P. This is called k-induction, and it combines well with BMC: we first perform BMC of depth k and if that fails to refute the formula, we try k-induction (the formulas generated are very similar to those for BMC), and if that fails, we repeat the process for $k+1$ ($k+1$-induction is stronger—proves more formulas—than k-induction). Subject to certain side conditions (for example, the initial k-sequence should be acyclic), k-induction is a *complete* method for finite-state systems. These results generalize from the finite- to infinite-state case when ICS is substituted for a SAT solver, and the method becomes complete for important classes of infinite-state systems, such as timed automata [4].

Our Infinite BMC procedure built on ICS has been extended to perform k-induction (with additional optimizations–e.g., requiring that only the first state in a sequence may be an initial state) and to strengthen invariants (using the heuristic described earlier). Standard examples such as the abstracted Futurebus and Illinois cache coherence protocols are verified in seconds by this method, and standard timed automata examples such as the Fischer protocol and train gate controller are verified in fractions of a second. These results suggest that ICS can be competitive with specialized systems operating in their own domains.

6 Conclusion

ICS packages a powerful and efficient set of deductive capabilities in the form of a C library that can easily be accessed by other applications. This makes deduction available as an *embedded* capability, whereas previously it was available only through theorem provers intended for standalone operation.

[2] For some examples, it is possible to compute a *completeness threshold*, such that failure to find a counterexample shorter than the threshold is sufficient for verification. However, for most examples in practice, it is either too expensive to compute the threshold, or its value is beyond the reach of BMC.

Powerful embedded deduction will allow many conventional tools to provide new capabilities, or more potent forms of existing capabilities, at little cost. For example, a compiler can perform truly accurate common subexpression detection by asserting the path predicates to ICS, then using its canonizer to compare subexpressions.

Simple formal analysis tools (e.g., completeness and consistency checkers for tabular specifications, test case generators, and bounded model checkers) can obtain most of their deductive support from ICS, with little deductive "glue" needed in the application.

We plan to enlarge the services provided by ICS so that even less deductive glue will be required in future. In particular, we intend to add quantifier elimination, rewriting (which will also perform definition expansion), and forward chaining (which is very effective for transitive relations). The quantified form of the combination of theories used in ICS is not decidable (e.g., quantified integer linear arithmetic—Presburger Arithmetic—becomes undecidable when uninterpreted function symbols are added), but the circumstances that trigger undecidability are sharply defined (and rare in practice) so that it is possible to decide a very large and useful fragment of the full theory. We expect that our methods will be heuristically effective on the undecidable fragment also, and on other undecidable extensions (e.g., nonlinear integer arithmetic).

Other planned enhancements include generation of concrete solutions to satisfiability problems (and hence concrete counterexamples to BMC problems), and generation of proof objects (independently checkable explanations for the decisions made by ICS). We expect that the latter will also improve the interaction between core ICS and its SAT solver, and thereby further increase the performance of full ICS.

ICS focuses on providing full automation for the cases where that is effective; we do not intend to extend ICS to a general theorem prover. However, just as our original decision procedures made it possible for PVS (and its NSA-sponsored predecessor EHDM) to have a different architecture and style of interaction than previous interactive theorem provers [10], so the increased capability of ICS will allow future systems to support new and more productive styles of human interaction. We intend to explore these opportunities in our research with future versions of PVS, and to assist NSA to do the same with its own systems.

ICS is freely available for noncommercial research purposes under license to SRI. Please visit its home page at `ics.csl.sri.com`.

Acknowledgments

We are grateful for the support and guidance provided by Bill Legato and Frank Rimlinger in tailoring ICS to applications of interest to NSA.

References

Papers on formal methods and automated verification by SRI authors can generally be located by visiting home pages or doing a search from `http://www.csl.sri.com/programs/formalmethods`.

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, Volume 1579 of Springer-Verlag *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999.

[2] David Cyrluk, Harald Rueß, and Oliver Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 60–71, Haifa, Israel, June 1997.

[3] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Presented at SAT 2002, accepted for journal publication, May 2002. Available at `http://www.csl.sri.com/users/demoura/sat02_journal.pdf`.

[4] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. Submitted for publication.

[5] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002.

[6] Jonathan Ford and Natarajan Shankar. Verifying Shostak. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 347–362, Copenhagen, Denmark, July 2002.

[7] Oliver Möller and Harald Rueß. Solving bit-vector equations. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, Volume 1522 of Springer-Verlag *Lecture Notes in Computer Science*, pages 36–48, Palo Alto, CA, November 1998.

[8] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[9] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[10] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[11] Sanjai Rayadurgam and Mats Heimdahl. Test-sequence generation from formal requirement models. In *High-Assurance Systems Engineering Symposium*, pages 23–31, IEEE Computer Society, Boca Raton, FL, October 2001.

[12] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, IEEE Computer Society, Boston, MA, July 2001.

[13] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, Volume 2378 of Springer-Verlag *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002.

[14] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[15] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 295—313, Copenhagen, Denmark, July 2002.

Bounded Model Checking and Induction: From Refutation to Verification ⋆

Leonardo de Moura, Harald Rueß, and Maria Sorea⋆⋆

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{demoura, ruess, sorea}@csl.sri.com
http://www.csl.sri.com/

Abstract. We explore the combination of bounded model checking and induction for proving safety properties of infinite-state systems. In particular, we define a general k-induction scheme and prove completeness thereof. A main characteristic of our methodology is that strengthened invariants are generated from failed k-induction proofs. This strengthening step requires quantifier-elimination, and we propose a *lazy* quantifier-elimination procedure, which delays expensive computations of disjunctive normal forms when possible. The effectiveness of induction based on bounded model checking and invariant strengthening is demonstrated using infinite-state systems ranging from communication protocols to timed automata and (linear) hybrid automata.

1 Introduction

Bounded model checking (BMC) [5, 4, 7] is often used for refutation, where one systematically searches for counterexamples whose length is bounded by some integer k. The bound k is increased until a bug is found, or some pre-computed *completeness threshold* is reached. Unfortunately, the computation of completeness thresholds is usually prohibitively expensive and these thresholds may be too large to effectively explore the associated bounded search space. In addition, such completeness thresholds do not exist for many infinite-state systems.

In deductive approaches to verification, the *invariance rule* is used for establishing invariance properties φ [11, 10, 13, 3]. This rule requires a property ψ which is stronger than φ and *inductive* in the sense that all initial states satisfy ψ, and ψ is preserved under each transition. Theoretically, the invariance rule is adequate for verifying a valid property of a system, but its application usually requires creativity in coming up with a sufficiently strong inductive invariant. It is also nontrivial to detect bugs from failed induction proofs.

In this paper, we explore the combination of BMC and induction based on the *k-induction* rule. This induction rule generalizes BMC in that it requires demonstrating

⋆ Funded by SRI International, by NSF Grant CCR-0082560, DARPA/AFRL-WPAFB Contract F33615-01-C-1908, and NASA Contract B09060051.

⋆⋆ Also affiliated with University of Ulm, Germany.

the invariance of φ in the first k states of any execution. Consequently, error traces of length k are detected. This induction rule also generalizes the usual invariance rule in that it requires showing that if φ holds in every state of every execution of length k, then every successor state also satisfies φ. In its pure form, however, k-induction does not require the invention of a strengthened inductive invariant. As in BMC, the bound k is increased until either a violation is detected in the first k states of an execution or the property at hand is shown to be k-inductive. In the ideal case of attempting to prove correctness of an inductive property, 1-induction suffices and iteration up to a, possibly large, complete threshold, as in BMC, is avoided. The k-induction rule is sound, but further conditions, such as the restriction to acyclic execution sequences, must be added to make k-induction complete even for finite-state systems [17].

One of our main contributions is the definition of a general k-induction rule and a corresponding completeness result. This induction rule is parameterized with respect to suitable notions of simulation. These simulation relations induce different notions of path *compression* in that an execution path is compressed if it does not contain two similar states. Many completeness results, such as k-induction for timed automata, follow by simply instantiating this general result with the simulation relation at hand. For general transition systems, we develop an *anytime* algorithm for approximating adequate simulation relations for k-induction.

Whenever k-induction fails to prove a property φ, there is a counterexample of length $k+1$ such that the first k states satisfy φ and the last state does not satisfy φ. If the first state of this trace is reachable, then φ is refuted. Otherwise, the counterexample is labeled *spurious*. By assuming the first state of this trace is unreachable, a spurious counterexample is used to automatically obtain a strengthened invariant. Many infinite-state systems can only be proven with k-induction enriched with invariant strengthening, whereas for finite systems the use of strengthening decreases the minimal k for which a k-induction proof succeeds.

Since our invariant strengthening procedure for k-induction heavily relies on eliminating existentially quantified state variables, we develop an effective quantifier elimination algorithm for this purpose. The main characteristic of this algorithm is that it avoids a potential exponential blowup in the initial computation of a disjunctive normal form whenever possible, and a constraint solver is used to identify relevant conjunctions. In this way the paradigm of *lazy* theorem proving, as developed by the authors for the ground case [7], is extended to first-order formulas.

The paper is organized as follows. Section 2 contains background material on encodings of transition systems in terms of logic formulas. In Section 3 we develop the notions of reverse and direct simulations together with an anytime algorithm for computing these relations. Reverse and direct simulations are used in Section 4 to state a generic k-induction principle and to provide sufficient conditions for the completeness of these inductions. Sections 5 and 6 discuss invariant strengthening and lazy quantifier elimination. Experimental results with k-induction and invariant strengthening for various infinite-state protocols, timed automata, and linear hybrid systems are summarized in Section 7. Comparisons to related work are in Section 8.

2 Background

Let $V := \{x_1, \ldots, x_n\}$ be a set of variables interpreted over nonempty domains $\mathcal{D}_1$ through $\mathcal{D}_n$, together with a type assignment τ such that $\tau(x_i) = \mathcal{D}_i$. For a set of typed variables V, a *variable assignment* is a function ν from variables $x \in V$ to an element of $\tau(x)$. The variables in $V := \{x_1, \ldots, x_n\}$ are also called *state variables*, and a *program state* is a variable assignment over V.

All the developments in this paper are parametric with respect to a given constraint theories $\mathcal{C}$, such as linear arithmetic or a theory of bitvectors. We assume a computable function for deciding satisfiability of a conjunction of constraints in $\mathcal{C}$. A set of *Boolean constraints*, $\mathsf{Bool}(\mathcal{C})$, includes all constraints in $\mathcal{C}$ and is closed under conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. Effective solvers for deciding the satisfiability problem in $\mathsf{Bool}(\mathcal{C})$ have been previously described [7, 6].

A tuple $\langle V, I, T\rangle$ is a *$\mathcal{C}$-program* over V, where interpretations of the typed variables V describe the set of states, $I \in \mathsf{Bool}(\mathcal{C}(V))$ is a predicate that describes the initial states, and $T \in \mathsf{Bool}(\mathcal{C}(V \cup V'))$ specifies the transition relation between current states and their successor states (V denotes the current state variables, while V' stands for the next state variables). The semantics of a program is given in terms of a *transition system* M in the usual way.

For a program $M = \langle V, I, T\rangle$, a sequence of states $\pi(s_0, s_1, \ldots, s_n)$ forms a *path* through M if $\bigwedge_{0 \le i < n} T(s_i, s_{i+1})$. A state s is *reachable* in M if there is a path $\pi(s_0, s_1, \ldots, s_{n-1}, s)$ through M and $I(s_0)$, and a state property $\varphi \in \mathcal{C}(V)$ is *invariant* in M iff $\varphi(s)$ holds for every reachable state s in M. A *counterexample* for a property φ is a path $\pi(s_0, \ldots, s_n)$ such that $I(s_0)$ and $\neg\varphi(s_n)$, and the length $len(\pi)$ of such a counterexample is given by the number of states in this path.

Typical programming constructs can be rewritten into the program syntax presented above. For example, Dijkstra's guarded commands are encoded in terms of a disjunction of conjunctions of guards $g(x_1, \ldots, x_n)$ and updates $x_i' = f_1(x_1, \ldots, x_n)$ for all variables x_i. Programs with external, non-deterministic inputs are defined by partitioning the set of variables into input variables, which are unconstrained, and the other state variables, whose next-state values are constrained by the transition relation.

Throughout this paper we use timed automata [2], which are state-transition graphs augmented with a finite set of real-valued clocks, as a prototypical class of infinite-state systems. Decidability of the model-checking problem for timed automata rests on the fact that the space of clock valuations is partitioned into finitely many clock regions. Two clock valuations v_1, v_2 that belong to the same region are (region) equivalent, denoted as $v_1 \sim_{TA} v_2$. This region equivalence is a *stable* quotient relation, that is, whenever $q \sim_{TA} u$ and $T(q, q')$, there exists a state u' such that $T(u, u')$ and $q' \sim_{TA} u'$ [2]. Encoding of timed automata in terms of logical programs with linear arithmetic constraints are described in [19]. In particular, program states consist of a location and nonnegative real interpretations of clocks. For timed automata we restrict ourselves to proving so-called clock constraints φ, such that $q \sim_{TA} u$ implies that $\varphi(q)$ iff $\varphi(u)$.

3 Direct and Reverse Simulation

The notions of direct and reverse simulation as developed here lay out the foundation for the completeness results in Section 4.

Definition 1 (Direct / Reverse Simulation). Let $M = \langle V, I, T\rangle$ be a program and φ a state formula over V. We define the functors F_d and F_r that map binary relations R over V in the following way.

$$F_d(R)(s_1, s_2) := \begin{cases} \textit{if } \neg\varphi(s_1) \textit{ then } \neg\varphi(s_2) \\ \textit{else } \forall s_1' \,.\, T(s_1, s_1') \Rightarrow \exists s_2' \,.\, R(s_1', s_2') \ \wedge\ T(s_2, s_2') \end{cases}$$

$$F_r(R)(s_1, s_2) := \begin{cases} \textit{if } I(s_1) \textit{ then } I(s_2) \\ \textit{else } \forall s_1' \,.\, T(s_1', s_1) \Rightarrow \exists s_2' \,.\, R(s_1', s_2') \ \wedge\ T(s_2', s_2) \end{cases}$$

A *direct simulation* over V with respect to φ is any binary relation $\preceq$ over V that satisfies $\preceq \subseteq\ F_d(\preceq)$. Similarly, a *reverse simulation* over V with respect to φ is any binary relation $\preceq$ over V that satisfies $\preceq \subseteq\ F_r(\preceq)$.

In contrast to reverse simulations, direct simulations depend on a state formula φ. Also, the definition of direct simulation is inspired by the notion of *stable* relations above. Direct (reverse) simulations are usually denoted by $\preceq_d$ ($\preceq_r$). The following direct and reverse simulations are used as running examples throughout the paper.

Example 1. The empty relation $a \preceq_\emptyset b := \textit{false}$ is a direct and a reverse simulation.

Example 2. Equality ($=$) between states is a direct and a reverse simulation.

Example 3. The relation $s_1 \preceq_I s_2 := I(s_1) \wedge I(s_2)$ is a reverse simulation, where I is the predicate for describing the set of initial states of the given program.

Example 4. Now, consider programs $\langle V, I, T\rangle$ with inputs such that $\textit{input}(x)$ holds iff x is an input variable. The relation

$$s_1 =_i s_2 := \textit{for all variables } x \in V \,.\, \textit{input}(x) \textit{ or } s_1(x) = s_2(x),$$

with $s(x)$ denoting the value of the variable x in the state s, is a reverse simulation, since the values of the input variables are not constrained by the predicate I and their next values are not constrained by T. Obviously, for transition systems with inputs, the relation $s_1 =_i s_2$ is weaker than $=$, and therefore gives rise to shorter paths.

Example 5. We now consider timed automata programs and clock constraints. The region equivalence $\sim_{TA}$, which give rise to finitely many clock regions, is stable, and therefore a direct simulation.

The notions of direct and reverse simulation are modular in the sense that the union of direct (reverse) simulations is also a direct (reverse) simulation.

Proposition 1 (Modularity). If $\preceq_1$ and $\preceq_2$ are direct (reverse) simulations, then $\preceq_1 \cup \preceq_2$ is also a direct (reverse) simulation.

This property follows directly from the definitions of direct (reverse) simulations in Definition 1 and from the monotonicity of the functors F_d and F_r. For example, the reverse simulations $\preceq_I$ and $=_i$ in Examples 3 and 4 may be combined to obtain a new reverse simulation.

Given a program $M = \langle V, I, T\rangle$ and a property φ, the associated *largest direct (reverse) simulation* relation $\preceq_D$ ($\preceq_R$) is obtained as the greatest fixpoint of the functor F_d (F_r) in Definition 1. These fixpoints exist, since F_d and F_r are monotonic. However, the fixpoint iterations are often prohibitively expensive, and a direct (reverse) simulation is only obtained on convergence of the iteration. The iteration in Proposition 2 provides a viable alternative in that a reverse (direct) simulation is refined to obtain a stronger reverse (direct) simulation. The proof of the proposition below follows from the definitions of reverse (direct) simulations, from the monotonicity of the functors F_r (F_d), and from modularity (Proposition 1).

Proposition 2 (Anytime Iteration). If $\preceq_r$ ($\preceq_d$) is a reverse (direct) simulation, then for all $n \geq 0$ the relation $\preceq_{r,n}$ ($\preceq_{d,n}$) is also a reverse (direct) simulation:

$$\begin{array}{ll} \preceq_{r,0} := \preceq_r & \preceq_{d,0} := \preceq_d \\ \preceq_{r,n} := \preceq_{r,n-1} \cup F_r(\preceq_{r,n-1}) & \preceq_{d,n} := \preceq_{d,n-1} \cup F_d(\preceq_{d,n-1}) \end{array}$$

Consequently, this iteration gives rise to an *anytime* algorithm for computing direct (reverse) simulations, and equality $=$, for example, may be used as seed, since it is both a direct and a reverse simulation (see Example 2). Also, quantifier elimination algorithms such as the one in Section 6 may be used in this iteration.

4 Completeness of k-Induction

Given the notions of direct and reverse simulations, we develop sufficient conditions for proving completeness of k-induction. These results are based on restricting paths to not contain states that are similar with respect to a given *direct* or *reverse* simulation. For direct (reverse) simulations we define a compressed path w.r.t. to the given direct (reverse) simulation as a path $\pi(s_0, s_1, \ldots, s_n)$ not containing any s_i, s_j with $j < i$ ($i < j$) such that s_i directly (reversely) simulates s_j.

Definition 2 (Path Compression).

– A path $\pi^{\preceq_d}(s_0, s_1, \ldots, s_n)$ is *compressed* w.r.t. the direct simulation $\preceq_d$ if:

$$\pi^{\preceq_d}(s_0, s_1, \ldots, s_n) := \pi(s_0, s_1, \ldots, s_n) \wedge \bigwedge_{0 \leq j < i \leq n} s_i \not\preceq_d s_j.$$

– A path $\pi^{\preceq_r}(s_0, s_1, \ldots, s_n)$ is *compressed* w.r.t. the reverse simulation $\preceq_r$ if:

$$\pi^{\preceq_r}(s_0, s_1, \ldots, s_n) := \pi(s_0, s_1, \ldots, s_n) \wedge \bigwedge_{0 \leq i < j \leq n} s_i \not\preceq_r s_j.$$

A path that is compressed with respect to the reverse and the direct simulations $\preceq_r$ and $\preceq_d$ is denoted by $\pi^{\preceq_{r,d}}$.

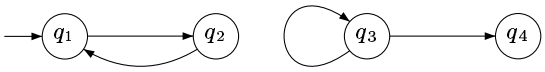

Fig. 1. Incompleteness of k-induction.

For example, a path $\pi(s_0, \ldots, s_n)$ is compressed w.r.t. the reverse simulation $(=)$ from Example 2 iff it is acyclic. Moreover, given the reverse simulation $\preceq_I$ from Example 3, a path $\pi(s_0, \ldots, s_n)$ is compressed w.r.t. $\preceq_I$ iff it contains at most one initial state. Obviously, for transition systems with inputs, the relation $(=_i)$ (see Example 4) is weaker than $(=)$, and therefore give rise to shorter compressed paths. We have collected all ingredients for defining k-induction for arbitrarily compressed paths.

Definition 3 (k-Induction). Let $M = \langle V, I, T \rangle$ be a program, k an integer, $\preceq_r$ a reverse simulation, and $\preceq_d$ a direct simulation. The induction scheme of depth k, $\mathrm{IND}^{\preceq_{r,d}}(k)$ allows one to deduce the invariance of φ in M if the following holds.

- $I(s_0) \wedge \pi^{\preceq_{r,d}}(s_0, \ldots, s_{k-1}) \rightarrow \varphi(s_0) \wedge \ldots \wedge \varphi(s_{k-1})$
- $\varphi(s_n) \wedge \ldots \wedge \varphi(s_{n+k-1}) \wedge \pi^{\preceq_{r,d}}(s_n, \ldots, s_{n+k}) \rightarrow \varphi(s_{n+k})$

For example, given the empty relationship $\preceq_\emptyset$ from Example 1, $\mathrm{IND}^{\preceq_\emptyset}$ reduces to the naive, incomplete k-induction on arbitrary paths. Consider, for example, the system in Figure 1 and a property φ, which is assumed to hold only in q_4. Now, the execution sequence $\underbrace{q_3 \rightsquigarrow q_3 \rightsquigarrow \ldots \rightsquigarrow q_3}_{k} \rightsquigarrow q_4$ is not k-inductive, but it is ruled out under the acyclic path restriction. The complete k-induction schemes in [17], which consider only acyclic paths and paths that only visit initial states once can be recovered by instantiating Definition 3 with the relations $(=)$ (Example 2) and $(\preceq_I)$ (Example 3), respectively. Since both $(=)$ and $(\preceq_I)$ are reverse simulations, an induction scheme restricted to acyclic paths visiting initial states at most once is obtained by modularity (Proposition 1).

Completeness of k-induction relies heavily on the notion of path compression. We now state the main lemma.

Lemma 1 (Compressing non-$\pi^{\preceq_{r,d}}$ paths). Let $\pi(s_0, \ldots, s_n)$ be a given path; then:

1. There exists a $\pi^{\preceq_r}$- compressed path $\pi^{\preceq_r}(q_0, \ldots, q_m)$, s.t. $q_m = s_n$ and $m \leq n$.
2. There exists a $\pi^{\preceq_d}$- compressed path $\pi^{\preceq_d}(q_0, \ldots, q_m)$, s.t. $q_0 = s_0$ and $m \leq n$.

Proofsketch. Assume a path $\pi(s_0, \ldots, s_n)$, which is not compressed w.r.t. $\preceq_r$. By Definition 1 it follows that there are states $s_i, s_j \in \pi(s_0, \ldots, s_n)$ such that $s_i \preceq_r s_j$, and $i < j$. We distinguish two cases. First, if s_i is an initial state, then so is s_j, and therefore a shorter path $\pi(s_j, \ldots, s_n)$ is obtained as a counterexample. Second, if s_i is not an initial state, then $s_i \neq s_0$, and there exists a s_{i-1} such that $T(s_{i-1}, s_i)$. Since $s_i \preceq_r s_j$ it follows by Definition 1 that there is a state s'_{i-1}, such that $s_{i-1} \preceq_r s'_{i-1}$ and $T(s'_{i-1}, s_j)$. If s_{i-1} is initial state, then so is s'_{i-1}, and since $i < j$ a shorter path $\pi^{\preceq_r}(s'_{i-1}, s_j, \ldots, s_n)$ is obtained. If s_{i-1} is not initial, by repeating the above argument

a shorter path is constructed. In both cases a shorter path is obtained, if such path is not a compressed path, then it is further reduced. The proof for $\pi^{\preceq_d}$- compressed paths works analogously.

$\text{IND}^{\preceq_{r,d}}(k)$ is *complete* if: φ is an invariant of M iff there is a k such that $\text{IND}^{\preceq_{r,d}}(k)(\varphi)$. Now, completeness of k-induction follows from the main lemma 1 above.

Theorem 1 (Completeness). $\text{IND}^{\preceq_{r,d}}(k)$ is a complete proof method iff there is an upper bound on the length of the paths $\pi^{\preceq_{r,d}}(s_0, \ldots, s_n)$.

Using the simulation from Example 2, Theorem 1 is instantiated to obtain the following complete k-induction for finite-state systems.

Corollary 1. Let M be a finite-state program over V and φ a state property in V; then $\text{IND}^{=}(k)$ induction is complete.

In general, k-induction for $(=)$ is not complete for infinite-state systems. Consider, for example, the program $M = \langle I, T\rangle$ over the integer state variable x with $I = (x = 0)$ and $T = (x' = x + 2)$, and the formula $x \neq 3$. Obviously, it is the case that $x \neq 3$ is invariant in M, but there exists no $k \in \mathbb{N}$ such that the property is proven by $\text{IND}^{=}(k)$. However, k-induction is complete for timed automata, since the equivalence relation $\sim_{TA}$ is a direct simulation (Example 5), and an upper bound on the length of the paths $\pi^{\sim_{TA}}(s_0, \ldots, s_n)$ is given by the number of clock regions.

Corollary 2. Let M be a timed automata program over the clock evaluations C and φ a clock constraint in C; then $\text{IND}^{\sim_{TA}}(k)$ induction is complete.

Similar results are obtained for other direct and reverse simulations and combinations thereof.

5 Invariant Strengthening

Whenever k-induction fails to prove a property φ, there is a counterexample $\pi = s_n, s_{n+1}, \ldots, s_{n+k}$ such that the first k states satisfy φ whereas the last state s_{n+k} does not satisfy this property. If s_n is indeed reachable, then φ is not invariant. Otherwise, the counterexample is labeled as *spurious* and it is inconclusive whether φ is invariant or not. However, by assuming s_n to be unreachable, such a spurious counterexample is used to obtain a strengthened invariant $\varphi \wedge \neg(s_n)$.

Consider, for example, the property $\neg(q_4)$ for the system in Figure 1. Induction of depth $k = 1$ fails, and the counterexample $q_3 \rightsquigarrow q_4$ is obtained. Now, $\neg(q_4)$ is strengthened to obtain $\neg(q_4) \wedge \neg(q_3)$, which is proven using 1-induction. More generally, whenever the induction step of $\text{IND}^{\preceq_{r,d}}(k)$ fails, the formula $Q(s_n, \ldots, s_{n+k}) := \varphi(s_n) \wedge \ldots \wedge \varphi(s_{n+k-1}) \wedge \pi^{\preceq_{r,d}}(s_n, \ldots, s_{n+k}) \wedge \neg\varphi(s_{n+k})$ is satisfiable, and each satisfying assignment describes a counterexample for the induction step. Thus, we define the predicate $U(s)$ for representing the set of possibly unreachable states, which may reach the bad state in k steps by means of a $\pi^{\preceq_{r,d}}$ path, $U(s) = \exists\, s_{n+1}, \ldots, s_{n+k}.Q(s, s_{n+1}, \ldots, s_{n+k})$. Now, φ is strengthened as $\varphi \wedge \neg U(s)$, and quantifier elimination is used for transforming this strengthened formula into an equivalent Boolean constraint formula. For the

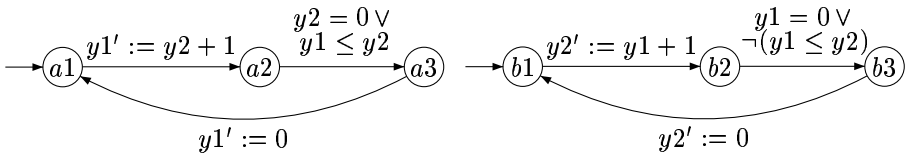

Fig. 2. Bakery Mutual Exclusion Protocol.

general case, we use the quantifier elimination procedure in Section 6. Notice, however, that for special cases such as guarded command languages, the quantifiers in $U(s)$ are eliminated using purely *syntactic* operations such as substitution, since all quantifications are over "next-state" variables x for which there are explicit solutions $f(.)$. An example might help to illustrate the combination of k-induction, strengthening, and quantifier elimination.

Example 6. Consider the usual stripped-down version of Lamport's Bakery protocol in Figure 2 with the initial value 0 for both counters $y1$ and $y2$ and the mutual exclusion property MX defined by $\neg(pc1 = a3 \wedge pc2 = b3)$. We apply 3-induction with the empty simulation relation $\preceq_\emptyset$. The base step holds and the induction step fails, thus we obtain

$$\begin{aligned} U(s_n) := \exists\, s_{n+1}, s_{n+2}, s_{n+3}.\ & MX(s_n) \wedge MX(s_{n+1}) \wedge \\ & MX(s_{n+2}) \wedge \pi^{\preceq_\emptyset}(s_n, s_{n+1}, s_{n+2}, s_{n+3}) \wedge \neg MX(s_{n+3}) \end{aligned}$$

with states s_i of the form $(pc1_i, y1_i, pc2_i, y2_i)$. Since the transitions of the Bakery protocol are in terms of guarded commands, simple substitution is used to obtain a quantifier-eliminated form, $R(s)$, defined as

$$R(s) := (pc1 = a1 \wedge pc2 = b2 \wedge y2 = 0) \vee (pc1 = a2 \wedge pc2 = b1 \wedge y1 = 0).$$

Now, the strengthened property $MX(s) \wedge \neg R(s)$ is proven using 3-induction.

6 Quantifier elimination

Given a quantified formula $\exists vars.\ \varphi$ with $\varphi \in \mathsf{Bool}(\mathcal{C})$, quantifier-elimination procedures usually work by transforming φ into disjunctive normal form (DNF) and distributing the existential quantifiers over disjunctions. Thus, one is left with eliminating quantifiers from a set of existentially quantified conjunctions of literals. We assume as given such a procedure *$\mathcal{C}$-qe*. The main drawback of these procedures is that there is a potential exponential blowup in the initial transformation to DNF and *$\mathcal{C}$-qe* might even return further disjunctions (as is the case for Presburger arithmetic); this problem has been addressed for the Boolean case by McMillan [14].

The quantifier elimination problem for invariant strengthening, as discussed in Section 5 allows for a purely syntactic quantifier elimination as long as we are restricting ourselves to guarded command programs. In these cases, *$\mathcal{C}$-qe* just applies the *substitution rule* ($x \notin vars(\psi)$)

$$(\exists x.(x = \psi) \wedge \varphi(x)) \text{ iff } \varphi(\psi);$$

```
procedure qe(vars, φ)
  ψ := false
  loop
    c := next-solution(φ)
    if c = false then return ψ
    c' := C-qe(vars, c)
    ψ := ψ ∨ c'
    φ := φ ∧ ¬c'
```

Fig. 3. Lazy Quantifier Elimination.

possibly followed by simplification. Quantifier elimination by substitution has already been used in the context of model checking, for example, by Coudert, Berthet, and Madre [15] and more recently by Williams, Biere, Clarke, Gupta [20], and Abdulla, Bjesse, Eén [1]. Another $\mathcal{C}$-*qe* function is used in McMillan's [14] quantifier elimination algorithm based on propositional SAT solving, in that his $\mathcal{C}$-*qe*$(vars, c)$ simply deletes the literals in c, which contain a variable in *vars*. In contrast, depending on the background theory, arbitrary complex quantifier elimination procedures, such as the ones for Presburger arithmetic or real-closed fields, can also be used here.

As motivated above, the initial DNF computation should usually be avoided when possible. Given a set of existentially quantified variables $vars$ and a quantifier-free formula φ in $\mathsf{Bool}(\mathcal{C})$, the algorithm $qe(vars, \varphi)$ in Figure 3 returns a formula in $\mathsf{Bool}(\mathcal{C})$ which is equivalent to $\exists vars.\ \varphi$. The procedure qe relies on a satisfiability solver for formulas $\varphi \in \mathsf{Bool}(\mathcal{C})$, which is assumed to enumerate representations of sets of satisfiable models in terms of conjunctions of literals in φ. Such a solver is described, for example, in [7, 6]. These solutions are supposed to be enumerated by successive calls to *next-solution* in Figure 3. Since there are only a finite number of solutions in terms of subsets of literals, the function qe is terminating. Moreover, minimal solutions or good over-approximations thereof, as produced by the lazy theorem proving algorithm [7, 6], accelerate convergence.

The variable c in Figure 3 stores the current solution obtained by *next-solution*, and the procedure $\mathcal{C}$-*qe* applies quantifier elimination for conjunction. In many cases, $\mathcal{C}$-*qe* just applies the *substitution rule* to remove quantified variables. In order to obtain the next set of solutions, we rule out the current solutions by updating φ with the value $\neg c'$ instead of $\neg c$, since $\neg c'$ is more restrictive.

Thus, the quantifier elimination procedure in Figure 3 avoids eager computation of a disjunctive normal form. Moreover, a solver for $\mathsf{Bool}(\mathcal{C})$ is used to guide the search for relevant "conjunctions" in φ. In this way, the qe algorithm extends the lazy theorem proving paradigm described in [7, 6] to the case of first-order reasoning.

Example 7. Consider

$$\begin{aligned}&\exists x_1, y_1\,((x_0 = 1 \vee x_0 = 3 \vee y_0 > 1) \wedge x_1 = x_0 - 1 \wedge y_1 = y_0 + 1)\\ \vee\quad &((x_0 = -1 \vee x_0 = -3) \wedge x_1 = x_0 + 2 \wedge y_1 = y_0 - 1)) \wedge x_1 < 0\end{aligned}$$

A first satisfiable conjunction of literals is obtained by, say

$$c := y_0 > 1 \wedge x_1 = x_0 - 1 \wedge y_1 = y_0 + 1 \wedge x_1 < 0.$$

Now, application of the substitution rule yields $c' := y_0 > 1 \wedge x_0 - 1 < 0$, and, after updating φ with $\neg c'$ a second solution is obtained as

$$c := x_0 = -3 \wedge x_1 = x_0 + 2 \wedge y_1 = y_0 - 1 \wedge x_1 < 0.$$

Again, applying the substitution rule, one gets $c' := x_0 = -3 \wedge x_0 + 2 < 0$, and, since there are no further solutions, the quantifier-eliminated formula is $(y_0 > 1 \wedge x_0 - 1 < 0) \vee (x_0 = -3 \wedge x_0 + 2 < 0)$.

7 Experiments

We describe some of our experiments with k-induction and invariant strengthening. Our benchmark examples include infinite-state systems such as communication protocols, timed automata and linear hybrid systems.[1] In particular, Table 1 contains experimental results for the Bakery protocol as described earlier, Simpson's protocol [18] to avoid interference between concurrent reads and writes in a fully asynchronous system, well-known timed automata benchmarks such as the train gate controller and Fischer's mutual exclusion protocol, and three linear hybrid automata benchmarks for water level monitoring, the leaking gas burner, and the multi-rate Fischer protocol. Timed automata and linear hybrid systems are encoded as in [19]. Starting with $k = 1$ we increase k until k-induction succeeds. We are using invariant strengthening only in cases where *syntactic* quantifier elimination based on substitution suffices. In particular, we do not use strengthening for the timed and hybrid automata examples, that is, *C-qe* tries to apply the substitution rule, if the resulting satisfiability problems for Boolean combinations of linear arithmetic constraints are solved using the lazy theorem proving algorithm described in [7] and implemented in the ICS decision procedures [9].

System Name	Proved with *k*	Time	Refinements
Bakery Protocol	3	0.21	1
Simpson Protocol	2	0.16	2
Train Gate Controller	5	0.52	0
Fischer Protocol	4	0.71	0
Water Level Monitor	1	0.08	0
Leaking Gas Burner	6	1.13	0
Multi Rate Fischer	4	0.84	0

Table 1. Results for *k-induction.* Timings are in seconds.

[1] These benchmarks are available at http://www.csl.sri.com/~demoura/cav03examples

The experimental results in Table 1 are obtained on a 2GHz Pentium-IV with 1Gb of memory. The second column in Table 1 lists the minimal k for which k-induction succeeds, the third column includes the total time (in seconds) needed for all inductions from 0 to k, and the fourth column the number of strengthenings. Timings do not include the one for quantifier elimination, since we restricted ourselves to syntactic quantifier elimination only. Notice that invariant strengthening is essential for the proofs of the Bakery protocol and Simpson's protocol, since *k-induction* alone does not succeed for any k.

Simpson's protocol for avoiding interference between concurrent reads and writes in a fully asynchronous system has also been studied using traditional model checking techniques. Using an explicit-state model checker, Rushby [16] demonstrates correctness of a finitary version of this potentially infinite-state problem. Whereas it took around 100 seconds for the model checker to verify this stripped-down problem, k-induction together with invariant strengthening proves the general problem in a fraction of a second. Moreover, other nontrivial problems such as correctness of Illinois and Futurebus cache coherence protocols, as given by [8], are easily established using 1-induction with only one round of strengthening.

8 Related Work

We restrict this comparison to work we think is most closely related to ours. Sheeran, Singh, and Stålmarck's [17] also use *k-induction*, but their approach is restricted to finite-state systems only. They consider k-induction restricted to acyclic paths and each path is constrained to contain at most one initial state. These inductions are simple instances of our general induction scheme based on reverse and direct simulations. Moreover, invariant strengthening is used here to decrease the minimal k for which k-induction succeeds.

Our path compression techniques can also be used to compute tight completeness thresholds for BMC. For example, a *compressed recurrence diameter* is defined as the smallest n such that $I(s_0) \wedge \pi^{\preceq_{r,d}}(s_0, \ldots, s_n)$ is unsatisfiable. Using equality ($=$) for the simulation relation, this formula is equivalent to the *recurrence diameter* in [4]. A tighter bound of the recurrence diameter, where values of input variables are ignored, is obtained by using the reverse simulation $=_i$. In this way, the results in [12] are obtained as specific instances in our general framework based on reverse and direct simulations. In addition, the *compressed diameter* is defined as the smallest n such that

$$I(s_0) \wedge \pi^{\preceq_{r,d}}(s_0, \ldots, s_n) \wedge \bigwedge_{i=0}^{n-1} \neg \pi_i^{\preceq_{r,d}}(s_0, s_i)$$

is unsatisfiable, where $\pi_i^{\preceq_{r,d}}(s_0, s_i) := \exists s_1, \ldots, s_{i-1}.\pi^{\preceq_{r,d}}(s_0, s_1, \ldots, s_{i-1}, s_i)$ holds if there is a relevant path from s_0 to s_i with i steps. Depending on the simulation relation, this compressed diameter yields tighter bounds for the completeness thresholds than the ones usually used in BMC [4].

9 Conclusion

We developed a general k-induction scheme based on the notion of reverse and direct simulation, and we studied completeness of these inductions. Although any k-induction proof can be reduced to a 1-induction proof with invariant strengthening, there are certain advantages of using k-induction. In particular, bugs of length k are detected in the initial step, and the number of strengthenings required to complete a proof is reduced signifi cantly. For example, a 1-induction proof of the Bakery protocol requires three successive strengthenings each of which produces 4 new clauses. There is, however, a clear trade-off between the additional cost of using k-induction and the number of strengthenings required in 1-induction, which needs to be studied further.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.
2. R. Alur. Timed automata. In *Computer-Aided Verification, CAV 1999*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22, 1999.
3. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.
4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zh. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579, 1999.
5. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
6. L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. *Annals of Mathematics and Artificial Intelligence*, 2002. Accepted for publication.
7. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 438–455. Springer-Verlag, July 27-30 2002.
8. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification (CAV'00)*, pages 53–68, 2000.
9. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.
10. S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, Mar. 1975.
11. S. M. Katz and Z. Manna. A heuristic approach to program verification. In N. J. Nilsson, editor, *Proceedings of the 3rd IJCAI*, pages 500–512, Stanford, CA, Aug. 1973. William Kaufmann.
12. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proceedings of VMCAI'03*, Jan. 2003.
13. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, Jan. 1995.
14. K. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer-Aided Verification, CAV 2002*, volume 2404 of *LNCS*. Springer-Verlag, 2002.

15. O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.
16. J. Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism. Technical report, CSL, SRI International, Menlo Park, Menlo Park, CA, July 2002.
17. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. *LNCS*, 1954:108, 2000.
18. H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, Jan. 1990.
19. M. Sorea. Bounded model checking for timed automata. In *Proceedings of MTCS 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*, 2002.
20. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*. Springer-Verlag, 2000.